Ethical hacking

Academic Year 2024-2025

Gabriel Rovesti

April 20, 2025

Contents

Chapter 1

Introduction to Ethical Hacking

1.1 What is Ethical Hacking?

Ethical hacking refers to the practice of deliberately probing computer systems, networks, and applications to identify security vulnerabilities that malicious hackers could exploit. Unlike malicious hacking, ethical hacking is performed with explicit permission from the system owner, with the goal of enhancing security rather than compromising it.

1.1.1 Types of Hackers

Hackers are broadly classified into three categories based on their intentions and methodologies:

- **Black Hat Hackers**: These individuals conduct cyberattacks for personal gain, revenge, or hacktivism. Their activities are illegal and unauthorized.
- **White Hat Hackers**: Also known as ethical hackers, they perform security assessments with permission, following legal and ethical guidelines. Their goal is to improve security posture.
- **Gray Hat Hackers**: These operate in the ambiguous middle ground. They may find vulnerabilities without permission but might disclose them responsibly rather than exploiting them.

1.1.2 Need for Ethical Hackers

Organizations require ethical hackers to:

- Identify vulnerabilities before malicious actors do

- Test security controls and mechanisms
- Validate compliance with security policies and regulations
- Assess the effectiveness of security training and awareness programs
- Provide input for security improvement initiatives

1.1.3 Red Team vs. Blue Team

In cybersecurity exercises and professional settings, security professionals are often divided into two teams:

- **Red Team**: A group of security experts who use their skills to attack a system, simulating real-world threats. Their goal is to find vulnerabilities that could be exploited by actual attackers.
- **Blue Team**: Security personnel responsible for defending against the Red Team's attacks. They monitor systems, detect intrusions, and respond to security incidents.

These teams help organizations:

- Identify misconfigurations and security gaps
- Improve detection capabilities and response times
- Create healthy competition among security staff
- Raise awareness about security risks
- Build security skills in a safe environment

1.2 Threat Modeling

Threat modeling is a systematic approach to identifying potential security threats, vulnerabilities, and attack vectors in a system. It involves analyzing a system from an adversary's perspective to identify and prioritize potential threats.

1.2.1 Approaches to Threat Modeling

Asset-Centric Approach

This approach focuses on identifying and protecting valuable assets:

- Create a comprehensive list of assets
- Document assets, components, and data flows
- For each element, identify potential threats

Attacker-Centric Approach

This approach concentrates on understanding potential attackers:

- Create a list of potential threat actors (considering motives, means, and opportunities)
- Develop a list of threats based on attacker profiles

Application-Centric Approach

This approach examines the application's architecture and functionality:

- Create a diagram of the application architecture
- List threats for each element (using frameworks like STRIDE or OWASP Top 10)
- Rank threats using a classification model

1.2.2 Threat Modeling Process

Step 1: Decompose the Application

- Understand how the application interacts with external entities
- Create use cases to understand how the application is used
- Identify entry points where attackers could interact with the application
- Identify assets that attackers would target
- Identify trust levels representing access rights granted to external entities

This information should be documented in a Threat Model document, which typically includes:

- Application name and version
- Description of the application
- Document owner
- Participants and reviewers

The model should also identify external dependencies, which are elements external to the application code but still within the organization's control, such as:

- Production environment requirements
- Server configurations
- Network topology
- Security controls

Step 2: Determine and Rank Threats

- Use a threat categorization methodology like STRIDE or the Application Security Frame (ASF)
- Identify potential threat targets from the attacker's perspective
- Classify threats into categories
- Determine the security risk for each threat using models like DREAD or likelihood-impact

The STRIDE threat categorization framework includes:

Threat Type	**Description**	**Security Control**
Spoofing	Accessing and using another user's credentials	Authentication
Tampering	Maliciously changing data in storage or transit	Integrity
Repudiation	Performing prohibited operations in a system that lacks traceability	Non-Repudiation
Information Disclosure	Unauthorized access to information	Confidentiality
Denial of Service	Preventing legitimate users from accessing services	Availability
Elevation of Privilege	Gaining unauthorized access to privileged resources	Authorization

Table 1.1: STRIDE Threat Categories and Security Controls

Step 3: Determine Countermeasures and Mitigation

After ranking threats, organizations can address risks through:

- **Accept**: Decide that the business impact is acceptable

- **Eliminate**: Remove components that enable the vulnerability
- **Mitigate**: Add controls that reduce the risk impact

1.2.3 Common Vulnerability Scoring System (CVSS)

The Common Vulnerability Scoring System (CVSS) is an industry standard for assessing the severity of security vulnerabilities. Scores range from 0 to 10, with 10 representing the most severe vulnerabilities. CVSS scores help organizations prioritize vulnerability remediation efforts.

Chapter 2

Network Security

2.1 Network Security Fundamentals

Network security encompasses the protection of networking infrastructure from unauthorized access, misuse, or theft. It involves creating a secure environment for devices, applications, and users to operate safely within a network.

2.1.1 TCP/IP Model

The TCP/IP model provides a framework for understanding network communications:

OSI Layer	**TCP/IP Layer**	**Standard Protocols**
Application Presentation Session	Application	DNS, HTTP
Transport	Transport	TCP, UDP
Network	Internet	IP
Data Link Physical	Network Access	ARP, Ethernet

Table 2.1: Comparison of OSI and TCP/IP Models

2.1.2 Network Structure

In a basic network, devices are identified by:

- **IP Address**: Assigned within the network (e.g., 192.168.2.1)
- **MAC Address**: Device-specific identifier (e.g., 00:25:96:FF:FE:12:34:56)

Machines connect to networks via Network Interface Cards (NICs), which are physical devices with associated MAC addresses. NICs serve as the physical and logical interface between machines and networks.

2.2 Packet Sniffing and Spoofing

2.2.1 Packet Sniffing

Packet sniffing is a technique where an attacker captures and analyzes network traffic to gather information. This can include:

- Network addresses and topologies
- Connected devices
- Running services and protocols
- Sensitive user information transmitted in plaintext

For sniffing to be effective, the attacker must be connected to the transmission medium where the target traffic flows.

How Packets Are Received

When a packet arrives at a NIC:

1. The NIC checks if the packet's destination MAC address matches its own
2. If matched, the packet is copied into the NIC's memory
3. The packet is transferred to the kernel through the link-level driver
4. The packet is processed by the protocol stack before being passed to user-space applications

Promiscuous and Monitor Modes

By default, NICs discard packets not addressed to them. However:

- **Promiscuous Mode**: When enabled, a NIC forwards all received packets to the kernel, regardless of the destination address. This allows capture of all traffic on a wired network segment.
- **Monitor Mode**: The wireless equivalent of promiscuous mode, enabling capture of all wireless frames within range, regardless of the network they belong to.

2.2.2 Packet Spoofing

Packet spoofing involves forging critical information in network packets, such as the source address. While normal socket programming allows control over limited header fields (usually just the destination), advanced techniques enable attackers to manipulate other fields, including source addresses.

Smurf Attack

A Smurf attack is an example of a spoofing attack that utilizes the Internet Control Message Protocol (ICMP):

1. The attacker sends a spoofed ICMP echo request (ping) packet
2. The source address is set to the victim's IP address
3. The destination is a broadcast address (e.g., 192.168.2.255)
4. All devices on the network respond to the ICMP request
5. All responses go to the victim, overwhelming their system

This is a type of reflected amplification attack, where a small amount of attacker bandwidth generates a much larger volume of traffic directed at the victim.

2.3 TCP/IP Protocol Attacks

2.3.1 Layer 4 Protocols

Transport layer protocols manage communication between applications:

- **Transmission Control Protocol (TCP)**
  - Connection-oriented protocol
  - Includes error recovery through acknowledgments
  - Ensures reliable, ordered delivery of data
- **User Datagram Protocol (UDP)**
  - Connectionless protocol
  - Lower overhead than TCP
  - No error recovery or guaranteed delivery

Both protocols use port numbers (16-bit unsigned integers, 0-65535) to identify communicating applications.

2.3.2 TCP Three-Way Handshake

TCP connections are established through a three-way handshake:

1. Client sends SYN packet with initial sequence number n
2. Server responds with SYN-ACK packet (acknowledgment n+1, sequence p)
3. Client sends ACK packet (acknowledgment p+1, sequence n+1)

After this process, a TCP connection is established, and data transfer can begin.

2.3.3 TCP Reset (RST)

The TCP RST (reset) packet is used to terminate existing TCP sessions, typically for error reporting. When a host receives an RST packet that appears to be from its peer, it immediately terminates the connection.

2.3.4 Attacks on TCP

SYN Flooding

SYN flooding exploits the TCP handshake process:

1. Attacker sends numerous SYN packets to the server
2. Server allocates resources for each potential connection
3. Attacker spoofs source addresses or never completes the handshake
4. Server's connection queue fills up, preventing legitimate connections

RST Attack

The RST attack disrupts established TCP connections:

1. Attacker monitors the network to learn connection details (addresses, ports, sequence numbers)
2. Attacker crafts a spoofed RST packet appearing to come from one communication endpoint
3. The receiving endpoint terminates the connection
4. Further packets from the legitimate sender are discarded

TCP Session Hijacking

TCP session hijacking allows attackers to take over existing connections:

1. Attacker monitors the connection to learn sequence numbers
2. Attacker sends a spoofed packet with the expected sequence number
3. The receiver accepts the attacker's packet as legitimate
4. The legitimate sender's packets get rejected (sequence mismatch)
5. Attacker effectively controls the session

2.3.5 Countermeasures to TCP Attacks

- Use random initial sequence numbers to prevent sequence prediction
- Implement secure alternatives like SSL/TLS for critical communications
- Employ encryption to prevent packet sniffing
- Deploy intrusion detection systems to identify spoofing attacks
- Use SYN cookies to protect against SYN flooding
- Implement connection rate limiting

2.4 Firewall Security

2.4.1 Firewalls

Firewalls are security systems that monitor and control incoming and outgoing network traffic based on predetermined security rules. They establish a barrier between trusted internal networks and untrusted external networks, such as the Internet.

2.4.2 Design Goals

Effective firewalls should:

- Ensure all traffic between internal and external networks passes through the firewall (achieved via physical connections)
- Implement security policies to determine which traffic is authorized
- Be immune to penetration, running on trusted platforms with secure operating systems

2.4.3 Capabilities of a Firewall

Firewalls provide several key security functions:

- Act as a choke point to prevent unauthorized access
- Prohibit vulnerable services from entering or leaving the network
- Protect against IP spoofing and routing attacks
- Provide auditing and alerting capabilities
- Serve as a platform for additional security functions

2.4.4 Demilitarized Zone (DMZ)

A DMZ is a network segment that connects an organization's untrusted external network (e.g., the Internet) to its trusted internal network. Servers that need to be accessible from outside the organization (web servers, email servers, etc.) are placed in the DMZ, allowing external access while maintaining a security barrier to the internal network.

2.4.5 Firewall Packet Filtering

Firewalls inspect packets and apply rules to determine whether to forward or discard them. The filtering process typically includes:

- Examining each incoming and outgoing packet
- Applying administrator-defined rules to determine validity
- Discarding non-conforming packets
- Organizing rules in tables for efficient processing

2.4.6 Firewall Policy

Firewall policies define how traffic is controlled based on:

- **Users**: Access based on who is trying to access resources
- **Services**: Access based on the type of service being requested
- **Direction**: Access based on the direction of traffic flow

Firewalls typically have three response options:

- **Accept**: Allow the traffic
- **Deny**: Block the traffic (silently drop)
- **Reject**: Block the traffic and notify the sender

2.4.7 Types of Firewalls

Firewalls can inspect traffic at various levels of the protocol stack, from network packets up to application-specific details. The depth of inspection depends on the organization's security requirements.

2.4.8 Characteristics for Filtering

Firewalls can filter based on:

- **IP address and protocol values**: Source/destination addresses, port numbers, direction of flow, protocol flags
- **Application protocols**: Application-level gateways monitor specific protocols
- **User identity**: Often used with authentication protocols like IPSec
- **Network activity**: Time-based or behavior-based filtering

2.4.9 Packet Filtering Firewall

These firewalls apply rules to each IP packet based on:

- Source IP address
- Destination IP address
- Source and destination port numbers
- IP protocol field
- Interface
- TCP flags (SYN, ACK, RST, etc.)

Packet filtering typically follows one of two default policies:

- **Default deny**: "That which is not expressly permitted is prohibited"
- **Default allow**: "That which is not expressly prohibited is permitted"

From a security perspective, default deny is generally preferred, as it follows the principle of least privilege.

2.4.10 Attacks on Packet Filtering Firewalls

IP Address Spoofing

An attacker outside the network spoofs an internal IP address, hoping to bypass security controls that trust internal traffic.

Countermeasure: Block incoming packets with source addresses from internal networks (ingress filtering).

Source Routing Attacks

The attacker specifies the route a packet should take, potentially bypassing security measures.

Countermeasure: Discard packets with source routing options enabled.

Tiny Fragments Attack

The attacker uses IP fragmentation to create very small packets, separating the TCP header information across multiple fragments to evade filtering rules.

Countermeasure: Enforce rules requiring the first fragment to contain a minimum amount of the transport header, or reassemble fragments before filtering.

Chapter 3

Web Security

3.1 Introduction to Web Security

Web security involves protecting websites and web applications from various security threats and vulnerabilities. As web applications become more interactive and process sensitive user data, they present an expanding attack surface for malicious actors.

3.2 SQL Injection

SQL Injection is a code injection technique that exploits vulnerabilities in web applications that interact with databases. It occurs when user input is not properly validated or sanitized before being used in SQL statements.

3.2.1 Web Applications Interaction with Database

Web applications typically:

1. Receive input from users through various interfaces
2. Construct SQL queries using this input
3. Send the queries to a database for execution
4. Process the results and present them to users

This interaction creates a potential attack surface if user input is not properly handled.

3.2.2 SQL Injection Mechanism

SQL injection occurs when input data is interpreted as part of an SQL command rather than as data. For example, consider this PHP code:

```
$input_uname = $_GET['username'];
$input_pwd = $_GET['Password'];
$hashed_pwd = sha1($input_pwd);

$sql = "SELECT id, name, eid FROM credential
        WHERE name= '$input_uname'
        AND Password='$hashed_pwd'";
```

The developer intends for users to provide legitimate username and password values. However, an attacker might input:

```
Username: admin' --
Password: anything
```

This would transform the SQL query to:

```
SELECT id, name, eid FROM credential
WHERE name= 'admin' --' AND Password='
   d3b07384d113edec49eaa6238ad5ff00'
```

The '–' introduces a comment in SQL, causing the database to ignore the password check, potentially allowing the attacker to log in as "admin" without knowing the password.

3.2.3 SQL Injection Attack Types

Attacks on SELECT Statements

These attacks aim to bypass authentication, extract data, or gather information about the database structure:

- **Authentication Bypass**: Using comments or logical operators to modify query conditions
- **UNION Attacks**: Combining results from the original query with results from an injected query
- **Blind SQL Injection**: Inferring information through boolean conditions when direct output is not visible

Attacks on UPDATE Statements

These attacks modify data in the database:

```
$sql = "UPDATE credential SET
        nickname='$input_nickname',
        email='$input_email',
        address='$input_address',
        Password='$hashed_pwd',
        PhoneNumber='$input_phonenumber'
        WHERE ID=$id;";
```

An attacker might input:

```
Email: attacker@evil.com', admin='1
```

This could modify the query to:

```
UPDATE credential SET
nickname='NickName',
email='attacker@evil.com', admin='1',
address='Address',
Password='5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8',
PhoneNumber='123-456-7890'
WHERE ID=1;
```

This might grant the attacker administrative privileges by setting an "admin" field to 1.

3.2.4 The Fundamental Cause

The root cause of SQL injection is mixing data and code. When user input (data) is incorporated directly into SQL statements (code) without proper separation or sanitization, the boundary between data and code can be compromised.

3.2.5 Countermeasures

Input Validation and Sanitization

Inspect and filter user-provided data to remove or escape characters that might be interpreted as code:

```
$email = $mysqli->real_escape_string($_GET['Email']);
$pwd = $mysqli->real_escape_string($_GET['Password']);
$sql = "SELECT * FROM users WHERE email='$email' AND
    password='$pwd'";
```

Functions like 'mysqli::real$_{e}scape_{s}tring()'$ $in PHP encode special characters that have meaning in SQ$

Prepared Statements and Parameterized Queries

The most effective solution is to separate code from data using prepared statements:

```
$conn = new mysqli("localhost", "root", "password", "
    database");
$sql = "SELECT Name, Salary, SSN FROM employee WHERE eid
    = ? AND password = ?";
if ($stmt = $conn->prepare($sql)) {
    $stmt->bind_param("ss", $eid, $pwd);
    $stmt->execute();
    $stmt->bind_result($name, $salary, $ssn);
    // ...
```

```
}
```

With prepared statements:

- SQL statement template is sent to the database separately from the data
- Database parses, compiles, and optimizes the SQL statement without executing it
- Data is later bound to the statement and sent via a separate channel
- Database clearly distinguishes between code and data
- Even if attackers include SQL syntax in their input, it's treated as data, not code

3.3 Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is one of the most common web vulnerabilities. It occurs when an application includes untrusted data in a web page without proper validation or escaping, allowing attackers to inject client-side scripts that execute in users' browsers.

3.3.1 XSS Types

Stored XSS

In stored XSS attacks:

1. Attacker submits malicious script through a form or API
2. The application stores the script in a database
3. When other users request content containing the stored script
4. The script is injected into the page and executed in users' browsers

Example scenario:

- A website allows users to submit comments on articles
- Attacker submits a comment containing JavaScript code: '<script>malicious code here</script>'
- The comment is stored in the database
- When other users view the page with the comment, the script executes in their browsers

Reflected XSS

In reflected XSS attacks:

1. Attacker creates a malicious link containing script code
2. Victim clicks the link, sending the script to the server
3. Server includes the script in its response without proper escaping
4. Script executes in the victim's browser

Example scenario:

- A search function displays the search query on the results page
- Attacker crafts a URL: 'example.com/search?q=<script>malicious code</script>'
- When the victim visits this URL, the server reflects the script in the response
- The script executes in the victim's browser

DOM-Based XSS

DOM-based XSS occurs when:

1. JavaScript code on the page uses data from an attacker-controllable source
2. This data is inserted into the DOM in an unsafe way
3. The attack is executed entirely on the client side

DOM-based XSS requires:

- A source: DOM object capable of storing attacker-controlled text (e.g., URL fragments, localStorage)
- A sink: DOM API capable of executing scripts (e.g., document.write, innerHTML)

Example scenario:

- A page uses JavaScript to read the URL hash fragment and display it:

```
1     const hash = document.location.hash;
2     document.write('Showing results for ' + hash);
3
```

- Attacker crafts a URL: 'example.com/page.html<script>alert(document.cookie)</script>'
- When the victim visits this URL, the script in the hash executes in their browser

3.3.2 Impact of XSS Attacks

XSS attacks can lead to:

- Session hijacking through cookie theft
- Credential theft through fake login forms
- Sensitive data exfiltration
- Malware distribution
- Website defacement
- Keylogging user input

3.3.3 Countermeasures

Input Validation

Validate user input on both client and server sides to ensure it conforms to expected formats and content.

Output Encoding

Escape special characters when rendering user-supplied content in HTML:

- Convert ‘<‘ to ‘<‘
- Convert ‘>‘ to ‘>‘
- Convert ‘"‘ to ‘"‘
- Convert ‘’‘ to ‘'‘
- Convert ‘‘ to ‘&‘

Content Security Policy (CSP)

Implement CSP headers to restrict which scripts can execute on a page:

```
Content-Security-Policy: script-src 'self' https://
    trusted-cdn.com
```

HttpOnly and Secure Cookie Flags

Set HttpOnly flag on cookies to prevent JavaScript access:

```
Set-Cookie: sessionId=abc123; HttpOnly; Secure
```

Context-Specific Escaping

Apply context-appropriate encoding based on where user data appears:

- HTML context: HTML entity encoding
- JavaScript context: JavaScript string escaping
- CSS context: CSS escaping
- URL context: URL encoding

Use Modern Frameworks

Many modern frameworks (React, Angular, Vue.js) automatically escape content by default, reducing XSS risks.

3.4 Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is an attack that forces authenticated users to execute unwanted actions on web applications where they're currently authenticated. The attack works by exploiting the trust a website has in a user's browser.

3.4.1 Cross-Site Requests and Their Problems

- **Same-site request**: When a page from a website sends an HTTP request back to the same website
- **Cross-site request**: When a request is sent from one website to a different website

The key issue is that when browsers send requests, they automatically attach cookies associated with the destination domain, regardless of where the request originated. This means servers cannot reliably distinguish between legitimate same-site requests and malicious cross-site requests.

3.4.2 CSRF Attack Mechanism

CSRF attacks require:

1. A target website with a state-changing function (e.g., fund transfer, password change)
2. A victim user who has an active, authenticated session on the target website
3. A malicious website under the attacker's control

The attack process:

1. Attacker crafts a webpage that generates a cross-site request to the target website
2. Attacker lures the victim to visit the malicious website
3. The victim's browser automatically sends the victim's cookies with the request
4. The target website processes the request as legitimate because it includes valid authentication cookies

3.4.3 CSRF Attacks on HTTP GET Services

For services using GET requests, attackers can exploit HTML elements that automatically trigger GET requests:

```
<img src="http://bank.example/transfer?to=attacker&amount
    =1000" width="1" height="1">
```

When the victim loads a page containing this code, their browser automatically sends the request to the bank website, including their authentication cookies.

3.4.4 CSRF Attacks on HTTP POST Services

For services using POST requests, attackers can use hidden forms with automatic submission:

```
<body onload="document.forms[0].submit()">
<form action="http://bank.example/transfer" method="POST"
    >
    <input type="hidden" name="to" value="attacker">
    <input type="hidden" name="amount" value="1000">
</form>
</body>
```

When the victim loads a page containing this code, the form is automatically submitted with their authentication cookies.

3.4.5 Fundamental Causes of CSRF

The core issue is that servers cannot distinguish between legitimate same-site requests and malicious cross-site requests. While browsers know the origin of requests, this information isn't reliably transmitted to servers.

3.4.6 Countermeasures

Referer Header

The Referer header identifies the page that generated a request. Servers can check this to verify requests come from their own pages. However, this approach is unreliable because:

- The header may be suppressed for privacy reasons
- It can be spoofed in some scenarios
- Users might disable it in their browsers

CSRF Tokens

The most effective countermeasure is to include a randomly generated token in each form:

```
<form action="/transfer" method="post">
    <input type="hidden" name="csrf_token" value="
   random_token_value">
    <input type="text" name="amount">
    <input type="text" name="to">
    <button type="submit">Transfer</button>
</form>
```

The server:

- Generates a unique token for each user session
- Includes the token in all forms
- Verifies the token on each submission
- Rejects requests with missing or invalid tokens

Cross-site requests cannot include the correct token due to the same-origin policy, which prevents websites from reading content from other origins.

Same-Site Cookies

Modern browsers support the SameSite cookie attribute, which controls when cookies are sent with cross-site requests:

- **Strict**: Cookies are never sent with cross-site requests
- **Lax**: Cookies are sent with cross-site GET requests that result in top-level navigation

- **None**: Cookies are sent with all cross-site requests (requires Secure attribute)

Setting cookies with the SameSite attribute provides strong protection against CSRF:

```
Set-Cookie: sessionId=abc123; SameSite=Strict; Secure
```

3.5 HTTP Request Smuggling

HTTP Request Smuggling is an attack technique that exploits inconsistencies in how front-end servers (such as proxies or load balancers) and back-end servers interpret HTTP requests.

3.5.1 Attack Mechanism

The attack is based on ambiguities in HTTP request delimiters:

- Different servers may interpret `Content-Length` and `Transfer-Encoding` headers differently
- The attacker sends a specially crafted request that is interpreted differently by front-end and back-end servers
- This discrepancy allows "smuggling" a request (or part of it) through the front-end, creating confusion in the back-end

3.5.2 Attack Variants

- **CL.TE**: Front-end uses Content-Length, back-end uses Transfer-Encoding
- **TE.CL**: Front-end uses Transfer-Encoding, back-end uses Content-Length
- **TE.TE**: Both use Transfer-Encoding but interpret it differently

3.5.3 Consequences

HTTP Request Smuggling attacks can lead to:

- Bypassing security controls in the front-end
- Poisoning web server caches
- Hijacking sessions of other users
- Bypassing XSS filters and other security controls

3.5.4 Countermeasures

- Properly configure front-end and back-end servers to ensure consistent interpretation
- Strictly validate incoming requests
- Regularly update web servers and proxies
- Implement web application firewalls (WAFs) with specific rules for this type of attack

Chapter 4

Software Security

4.1 Memory Layout and Stack

Understanding memory organization is crucial for understanding software vulnerabilities. Modern operating systems organize process memory into several segments:

- **Text Segment (.text)**: Contains executable code
- **Data Segment (.data)**: Contains initialized global variables
- **BSS Segment (.bss)**: Contains uninitialized global variables
- **Heap**: Dynamic memory allocation region (grows upward)
- **Stack**: Temporary storage for function calls (grows downward)

4.1.1 Function Call Stack

The stack is a crucial data structure that manages function calls and local variables:

- Each function call creates a stack frame
- Stack frames contain return addresses, saved base pointers, function arguments, and local variables
- The stack grows from higher memory addresses to lower memory addresses

Key registers in x86 architecture:

- **EBP/RBP (Base Pointer)**: Points to the base of the current stack frame

- **ESP/RSP (Stack Pointer)**: Points to the top of the stack
- **EIP/RIP (Instruction Pointer)**: Points to the next instruction to execute

4.1.2 Stack Frame Structure

A typical stack frame in x86 architecture is organized as follows (from higher to lower addresses):

- Function arguments (EBP+8, EBP+12, etc.)
- Return address (EBP+4)
- Saved EBP value (EBP+0)
- Local variables (EBP-4, EBP-8, etc.)

When a function is called:

1. The caller pushes arguments onto the stack (in reverse order)
2. The CALL instruction pushes the return address onto the stack
3. The function prologue:
    - Pushes the current EBP value onto the stack
    - Sets EBP to the current ESP value
    - Decrements ESP to allocate space for local variables

When a function returns:

1. The function epilogue:
    - Sets ESP to EBP, deallocating local variables
    - Pops the saved EBP value from the stack
2. The RET instruction pops the return address and jumps to it

4.2 Buffer Overflow Vulnerability

4.2.1 Understanding Buffer Overflows

A buffer overflow occurs when a program writes data beyond the allocated memory boundaries of a buffer. This typically happens in languages like C and C++ that don't perform automatic bounds checking.

Consider this vulnerable function:

```c
int check_authentication() {
    int auth_flag = 0;
    char password_buffer[16];
    printf("Enter password: ");
    scanf("%s", password_buffer);
    /* Check password and set auth_flag to 1 if correct
    */
    return auth_flag;
}
```

The problem is that `scanf("%s", password_buffer)` doesn't limit the input length. If a user enters more than 16 characters:

1. The input overflows the `password_buffer`
2. Adjacent memory is overwritten, potentially including the `auth_flag` variable
3. This could allow authentication to be bypassed without the correct password

In more severe cases, an attacker could overwrite the return address, redirecting execution to malicious code.

4.2.2 Exploitation of Buffer Overflows

Buffer overflow exploitation typically involves:

1. Identifying a buffer overflow vulnerability
2. Crafting input that overflows the buffer and overwrites the return address
3. Directing the return address to attacker-controlled code

For successful exploitation, attackers need to:

- Determine the offset between the buffer base and the return address
- Find a suitable location for shellcode
- Ensure the new return address doesn't contain null bytes (which would terminate string operations)

4.2.3 Countermeasures

Developer Approaches

- Check data length before copying
- Use safe functions that limit copy operations:
  - `strncpy()` instead of `strcpy()`
  - `strncat()` instead of `strcat()`
  - `snprintf()` instead of `sprintf()`
  - `fgets()` instead of `gets()`
- Use safer libraries (e.g., libsafe)
- Use memory-safe languages (e.g., Java, Python, Rust)

Compiler Approaches

- **Stack Canaries**: Random values placed between buffers and control data. Before returning from a function, the canary value is checked. If it has been modified, the program terminates.

```
1     int secret = random;
2     void foo(char *str) {
3         int guard;
4         guard = secret;
5         char buffer[12];
6         strcpy(buffer, str);
7         if (guard == secret)
8             return;
9         else
10            exit(1);
11    }
12
```

- **Address Space Layout Randomization (ASLR)**: Randomly arranges the address space positions of key program segments, making it difficult for attackers to predict target addresses.
- **Non-executable Stack/Heap**: Marks memory regions as non-executable, preventing direct execution of injected code.

Operating System Approaches

- Memory protection mechanisms
- Process isolation

- Privilege separation
- Sandboxing

4.3 Return-to-libc Attack

4.3.1 Defeating Non-executable Stack

The non-executable stack countermeasure prevents direct execution of code injected onto the stack. Return-to-libc attacks bypass this protection by reusing existing code in the program or libraries.

4.3.2 Arbitrary Code Execution

Arbitrary Code Execution (ACE) refers to an attacker's ability to execute arbitrary commands on a target system. This can be achieved through:

- **Code injection**: Inserting malicious code into a program's memory
- **Code reuse**: Redirecting execution to existing code with malicious intent

4.3.3 Code Reuse Attacks

Code reuse attacks leverage code already present in memory:

- **Return-to-libc**: Redirecting execution to functions in the C standard library
- **Return-Oriented Programming (ROP)**: Chaining small code sequences ending with return instructions
- **Jump-Oriented Programming (JOP)**: Using jump instructions instead of returns

4.3.4 Return-to-libc Mechanism

A return-to-libc attack involves:

1. Exploiting a buffer overflow to overwrite the return address
2. Setting the return address to point to a useful function in the C library (often `system()`)
3. Providing arguments for that function (e.g., the string "/bin/sh" to spawn a shell)

The attack requires several key elements:

- Address of the `system()` function
- Address of a string like "/bin/sh"
- Proper stack arrangement to pass arguments to `system()`

4.3.5 Finding Function Addresses

Attackers can use debugging tools to find the address of functions like `system()`:

```
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7ecffb0 <system>
```

4.3.6 Finding String Arguments

Attackers can locate or create suitable string arguments:

- Search for existing strings in the program
- Place strings in environment variables
- Inject strings through input

4.3.7 Return-Oriented Programming (ROP)

ROP extends the return-to-libc concept by chaining together small code sequences (gadgets) that end with return instructions. This technique allows for more complex operations without introducing new code.

A ROP chain might:

1. Pop values into registers
2. Perform memory operations
3. Call specific functions
4. Chain multiple operations together

ROP gadgets can be identified using specialized tools:

- Ropper
- ROPgadget
- Pwntools

4.4 Format String Vulnerability

4.4.1 Format Strings

Format strings are used by functions like `printf()` to specify how to format output. A format string vulnerability occurs when user input is used directly as a format string.

```
int printf(const char *format, ...);
```

Format strings contain specifiers (beginning with %) that are replaced with values from additional arguments:

```
printf("Value: %d, String: %s", 42, "hello");
```

4.4.2 How printf() Works

The `printf()` function:

1. Scans the format string character by character
2. When it encounters a format specifier, it retrieves the corresponding argument
3. Formats the argument according to the specifier and adds it to the output

Arguments are accessed using the `va_list` mechanism:

```
void myprint(char *format, ...) {
    va_list ap;
    va_start(ap, format);
    // Process format string and retrieve arguments using
    va_arg(ap, type)
    va_end(ap);
}
```

4.4.3 Missing Arguments

A key vulnerability arises from the fact that `printf()` doesn't know how many arguments to expect. If format specifiers outnumber the provided arguments, `printf()` continues fetching values from the stack, potentially exposing sensitive information.

```
printf("%d %d %d %d", 10, 20);  // Missing arguments for
   last two %d
```

In this case, `printf()` will read values from the stack for the third and fourth %d, potentially revealing stack contents.

4.4.4 Format String Vulnerability

Format string vulnerabilities arise when user input is directly used as a format string:

```
// Vulnerable code
char user_input[100];
scanf("%s", user_input);
printf(user_input);  // User input used as format string
```

If a user enters a string containing format specifiers (e.g., "%x %x %x"), the `printf()` function will interpret these as format specifiers and attempt to retrieve arguments from the stack.

4.4.5 Exploitation Techniques

Information Disclosure

Attackers can use format string vulnerabilities to read data from the stack:

- `%x`: Prints values from the stack as hexadecimal
- `%s`: Treats values from the stack as pointers and prints the strings they point to

Memory Write

The `%n` specifier writes the number of characters output so far to the address specified by the corresponding argument. This can be used to write arbitrary values to arbitrary memory locations:

```
int count = 0;
printf("Hello %n World", &count);  // count becomes 6
```

By controlling the format string and manipulating the stack, attackers can use `%n` to overwrite critical memory, such as:

- Function pointers
- Return addresses
- Global Offset Table (GOT) entries

Width modifiers can be used to control the value written:

```
printf("%10000x%n", 1, &target);  // Writes 10001 to
    target
```

4.4.6 Countermeasures

Developer Countermeasures

Never use untrusted input as format strings. Always use a static format string:

```
// Vulnerable:
printf(user_input);

// Secure:
printf("%s", user_input);
```

Compiler Countermeasures

Modern compilers can detect potential format string vulnerabilities:

```
$ gcc -Wformat -Werror=format-security program.c
```

The `-Wformat=2` option enables additional warnings for format string issues.

System Countermeasures

- **Address Space Layout Randomization (ASLR)**: Makes it harder for attackers to predict memory addresses
- **Position Independent Executables (PIE)**: Work with ASLR to randomize the location of the executable code
- **FormatGuard**: A modified version of the C library that counts format specifiers and arguments

4.5 Shellcode

4.5.1 What is Shellcode?

Shellcode is a small piece of code used as the payload in the exploitation of software vulnerabilities. It derives its name from the fact that it traditionally spawns a command shell, though modern shellcode can perform many different tasks.

4.5.2 Creating Shellcode from C Code

To understand shellcode, let's start with a simple C program that spawns a shell:

```c
#include <stddef.h>
void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

This program uses the `execve()` system call to execute `/bin/sh`. To convert this to shellcode, we need to understand how `execve()` is called at the assembly level.

4.5.3 System Call Execution

In x86 Linux, system calls are invoked using the `int 0x80` instruction, with registers set to specific values:

- `eax`: System call number (11 for `execve()`)
- `ebx`: First argument (pointer to the filename)
- `ecx`: Second argument (pointer to the argument array)
- `edx`: Third argument (pointer to the environment array)

4.5.4 Shellcode Example

A basic shellcode to spawn a shell might look like this:

```
; Zero out eax
xorl %eax, %eax
; Push NULL onto the stack
pushl %eax
; Push "//sh" onto the stack (padded to 4 bytes)
pushl $0x68732f2f
; Push "/bin" onto the stack
pushl $0x6e69622f
; Store pointer to "/bin//sh" in ebx
movl %esp, %ebx
; Push NULL onto the stack
pushl %eax
; Push pointer to "/bin//sh" onto the stack
pushl %ebx
; Store pointer to the argument array in ecx
movl %esp, %ecx
; Zero out edx (environment pointer)
xorl %edx, %edx
; Load execve system call number (11) into al
movb $0x0b, %al
; Execute the system call
int $0x80
```

When converted to bytes, this becomes:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\
    xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80
```

4.5.5 Shellcode Characteristics

Effective shellcode typically has these characteristics:

- **Position-independence**: Functions without knowing its memory location
- **Null-free**: Contains no null bytes, which would terminate string operations
- **Compactness**: Small size to fit within limited buffer space
- **Stealth**: Designed to evade detection by security mechanisms

4.5.6 Advanced Shellcode Techniques

- **Encoder/Decoder Shellcode**: Encodes the payload to avoid detection, then decodes it at runtime
- **Staged Shellcode**: Initial small payload downloads or assembles a larger payload
- **Egg-hunting**: Searches memory for a larger payload when buffer space is limited
- **Socket Reuse**: Reuses existing network connections instead of creating new ones

4.6 Race Condition Vulnerability

Race conditions represent a class of vulnerabilities that occur when system behavior depends on the sequence or timing of uncontrollable events.

4.6.1 Fundamentals of Race Conditions

- Occur in concurrent or multi-threaded systems
- Depend on the order in which threads or processes execute
- Are difficult to detect because they are often not deterministically reproducible

4.6.2 Time-of-Check to Time-of-Use (TOCTOU)

TOCTOU is a particular category of race condition:

1. A program checks a condition (time-of-check)
2. There is a time interval before the program uses the result of the check
3. During this interval, the condition may change
4. The program acts based on information that is no longer valid (time-of-use)

```
// Vulnerable example
if (access("file.txt", W_OK) == 0) {  // Check
    // Possible state change here (race window)
    fd = open("file.txt", O_WRONLY);  // Use
}
```

4.6.3 Examples in Real Contexts

- **File systems**: Checking file permissions before access
- **Atomic operations**: Incrementing counters or operations on shared flags
- **Web applications**: Checking and updating records in databases
- **Operating systems**: Race conditions in drivers or privileged operations

4.6.4 Exploitation Techniques

- **Stress testing**: Repeated parallel execution to increase the probability of hitting the race condition
- **Time manipulation**: Artificially slowing down the execution of the target system
- **Forced interruption**: Forcing process interruption during critical windows

4.6.5 Countermeasures

- **Atomic operations**: Using synchronization primitives (mutexes, semaphores)
- **Transactional patterns**: Adopting approaches like "compare-and-swap"

- **Principle of least privilege**: Reducing permissions to minimize impact
- **File descriptors**: Maintaining open references to avoid changes
- **Canonical paths**: Fully resolving paths before use

Chapter 5

Blockchain Security

5.1 Distributed Ledger Technology

5.1.1 What is a Distributed Ledger Technology?

A Distributed Ledger Technology (DLT) is a decentralized replicated record of data managed and maintained by multiple untrusted entities without the need for a central regulatory authority. Key characteristics include:

- Decentralization: No single controlling entity
- Replication: Each participant maintains a copy of the ledger
- Consensus: Participants agree on ledger state without trusting each other
- Immutability: Once recorded, data is difficult to change

5.1.2 Centralized vs. Decentralized Systems

Centralized Systems

- Single point of control and failure
- Hierarchical structure
- Dependent on central authority
- Potentially faster and more efficient
- Examples: Traditional banking systems, centralized databases

Decentralized Systems

- Distributed control
- No single point of failure
- Peer-to-peer architecture
- Resistant to censorship
- May sacrifice some efficiency for security and resilience
- Examples: Blockchain networks, P2P file sharing

5.1.3 Replicated Record of Data

In a DLT:

- Data is organized as a list of records (transactions)
- Each node replicates this list of records
- Each record represents a transaction (e.g., payment from one user to another)
- The latest record defines the current state of the system
- The system can store various types of data:
  - Financial values
  - Smart contracts
  - IoT sensor data
  - Intellectual property rights

5.1.4 Management and Maintenance

- Each node has its own copy of the ledger
- Each node works to update its own copy
- Nodes work for the benefit of the distributed ledger
- Copies may differ temporarily, but consensus mechanisms ensure that the majority view prevails

5.1.5 Untrusted Members

A key feature of DLTs is their ability to function despite mutual distrust among participants:

- Nodes do not inherently trust other nodes in the network
- Each node directly communicates with peer nodes
- Each node has equal rights to propose modifications
- Immutability is ensured through append-only policies and cryptographic linking

5.2 Blockchain

5.2.1 What is Blockchain?

Blockchain is a specific type of DLT where data is organized into blocks that are cryptographically linked together in a chain. Key features that distinguish blockchain from other DLTs:

- Data is grouped into blocks rather than individual transactions
- Each block contains multiple transactions
- Blocks are cryptographically linked to previous blocks
- This linkage creates a tamper-evident chain

5.2.2 Block Structure

A typical blockchain block contains:

- Block header
  - Previous block hash (creates the chain)
  - Merkle root of transactions
  - Timestamp
  - Nonce (used in proof-of-work)
  - Difficulty target
- List of transactions
- Metadata

5.3 Smart Contracts

5.3.1 What are Smart Contracts?

Smart contracts are self-executing programs stored on a blockchain that run when predetermined conditions are met. They automate the execution of agreements so that all participants can be immediately certain of the outcome without an intermediary.

Key characteristics:

- Autonomous execution based on predefined conditions
- Decentralized operation without intermediaries
- Transparent and verifiable on the blockchain
- Immutable once deployed

5.3.2 Smart Contract Structure

In Ethereum, smart contracts are typically written in Solidity. A basic contract structure includes:

```
pragma solidity ^0.8.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Components:

- Pragma directive: Specifies the compiler version
- Contract declaration: Defines the contract name and body
- State variables: Store the contract's state on the blockchain
- Functions: Implement the contract's logic

5.3.3 Smart Contract Execution

Smart contracts are executed by the Ethereum Virtual Machine (EVM):

1. Contract code is compiled to EVM bytecode
2. Bytecode is stored in a transaction and deployed to the blockchain
3. To interact with a contract, users send transactions to the contract address
4. The transaction data field specifies which function to call and with what parameters
5. Miners execute the contract code and update the blockchain state

5.3.4 Smart Contract Security Vulnerabilities

Smart contracts are susceptible to various security vulnerabilities:

Reentrancy Attack

- Allows an attacker to recursively call a function before the first execution completes
- Led to the famous DAO hack in 2016, resulting in $50 million loss

```
// Contract vulnerable to reentrancy
function withdraw(uint amount) public {
    require(balances[msg.sender] >= amount);

    // Sending ETH - can be exploited to call withdraw() again
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success);

    // This line executes after the external call
    balances[msg.sender] -= amount;
}
```

```
// Secure version
function withdraw(uint amount) public {
    require(balances[msg.sender] >= amount);

    // First update the state
    balances[msg.sender] -= amount;

    // Then send ETH
```

```
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success);
}
```

Integer Overflow/Underflow

- In Solidity versions prior to 0.8.0, arithmetic operations did not prevent overflow/underflow
- Can lead to unexpected behaviors in token calculations, balances, etc.

```
// Vulnerable in Solidity < 0.8.0
function transfer(address to, uint256 amount) public {
    require(balances[msg.sender] >= amount);
    balances[msg.sender] -= amount;
    balances[to] += amount;  // Possible overflow
}
```

Race Conditions in Smart Contracts

Despite the sequential nature of blockchain execution, specific race conditions exist:

- **Front-running**: Miners or other actors can observe pending transactions and insert their own with higher gas prices
- **Transaction ordering**: The order of transactions in a block can be manipulated by miners

```
// Vulnerable to front-running
function claimReward(bytes32 solution) public {
    require(solutions[solution] == false);
    require(isValidSolution(solution));

    solutions[solution] = true;
    msg.sender.transfer(1 ether);  // Reward
}
```

An attacker can observe this transaction in the mempool, copy the solution, and send their own transaction with a higher gas price to steal the reward.

Front-Running Countermeasures

- **Commit-reveal**: Two-phase scheme where users first submit a hash of their solution, then reveal it
- **Batch auction**: All transactions in a specific period participate equally
- **Submarine sends**: Temporarily hiding the recipient address

```
// Commit-reveal approach
function commit(bytes32 solutionHash) public {
    commits[msg.sender] = solutionHash;
    commitTimes[msg.sender] = block.timestamp;
}

function reveal(string memory solution) public {
    bytes32 solutionHash = keccak256(abi.encodePacked(
   solution, msg.sender));
    require(commits[msg.sender] == solutionHash);
    require(block.timestamp >= commitTimes[msg.sender] +
   24 hours);

    require(isValidSolution(solution));
    require(!claimed[solution]);

    claimed[solution] = true;
    msg.sender.transfer(1 ether);
}
```

Chapter 6

Reverse Engineering

6.1 Introduction to Reverse Engineering

6.1.1 What is Reverse Engineering?

Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction. In cybersecurity, it typically involves examining compiled software to understand its functionality, structure, and behavior.

6.1.2 Why Reverse Engineering is Important

Reverse engineering serves several purposes in cybersecurity:

- Vulnerability assessment
- Malware analysis
- Penetration testing ("pwning")
- Algorithm analysis
- Interoperability (e.g., SMB/Samba, Windows/Wine)
- Analyzing embedded devices
- Understanding proprietary formats and protocols

6.1.3 Software Lifecycle and Reverse Engineering

Understanding the software lifecycle helps contextualize reverse engineering:

1. Source code is written by developers
2. Compiler translates source code to object code

3. Linker combines object code with libraries
4. Loader places the executable in memory
5. Dynamic linker resolves runtime dependencies

Reverse engineering attempts to recover higher-level information from the compiled code, working against the information loss that occurs during compilation.

6.1.4 Executable File Formats

Operating systems use specific formats for executable files:

- ELF (Executable and Linkable Format): Used in Unix-like systems
- PE (Portable Executable): Used in Windows
- Mach-O: Used in macOS and iOS

These formats define how programs are loaded into memory and executed.

6.1.5 Executable and Linkable Format (ELF)

ELF is the standard binary format for executables, object code, shared libraries, and core dumps in Unix-like systems. Its design features include:

- Flexibility and extensibility
- Cross-platform compatibility
- Support for various processor architectures

ELF files contain:

- Program headers: Describe segments (virtual memory mappings)
- Section headers: Describe sections and how to load them into segments
- Relocation information: For connecting symbolic references with definitions

6.2 Reverse Engineering Techniques

6.2.1 Static Analysis

Static analysis examines the program without executing it:

- **Disassembly**: Converting machine code to assembly language
  - Tools: objdump, IDA Pro, Ghidra, Radare2, Binary Ninja
- **Decompilation**: Attempting to recover higher-level code
  - Tools: Ghidra, Hex-Rays Decompiler
- **Binary analysis**: Examining file format, headers, and symbols
  - Tools: file, readelf, objdump, hexedit
- **Abstract interpretation**: Approximating program behavior without execution
- **Symbolic execution**: Exploring multiple execution paths simultaneously

6.2.2 Dynamic Analysis

Dynamic analysis involves running the program and observing its behavior:

- **Debugging**: Executing code step-by-step with full visibility
  - Tools: gdb, WinDbg, OllyDbg, Immunity Debugger
- **Tracing**: Recording program execution
  - Tools: strace, ltrace, ftrace
- **Dynamic binary instrumentation**: Inserting instrumentation code at runtime
  - Tools: Pin, DynamoRIO, Valgrind
- **Sandboxing**: Running code in isolated environments to observe behavior
  - Tools: Cuckoo Sandbox, any virtual machine

6.3 Assembly Language Basics

6.3.1 x86 and x86_64 Architecture

The x86 architecture is a CISC (Complex Instruction Set Computer) architecture initially developed by Intel. Key points:

- x86 refers to 32-bit architecture
- x86_64 (or AMD64) refers to 64-bit extension
- Both are widely used in personal computers and servers

6.3.2 Registers

Registers are small, fast storage locations within the processor:

64-bit Register	32-bit Register	16-bit Register	8-bit Register
RAX	EAX	AX	AH/AL
RBX	EBX	BX	BH/BL
RCX	ECX	CX	CH/CL
RDX	EDX	DX	DH/DL
RSI	ESI	SI	SIL
RDI	EDI	DI	DIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8-R15	R8D-R15D	R8W-R15W	R8B-R15B
RIP	EIP	IP	-

Table 6.1: x86_64 Registers

Special-purpose registers:

- **RSP/ESP**: Stack Pointer - points to the top of the stack
- **RBP/EBP**: Base Pointer - references function parameters and local variables
- **RIP/EIP**: Instruction Pointer - points to the next instruction to execute

6.3.3 Common x86 Instructions

Data Movement

- **MOV destination, source**: Copies data
  - `MOV EAX, 42` - Loads immediate value 42 into EAX

  - `MOV EAX, EBX` - Copies value from EBX to EAX
  - `MOV EAX, [EBX]` - Loads value from memory address in EBX to EAX

- **LEA destination, source**: Load Effective Address
  - `LEA ECX, [EAX+3]` - Calculates EAX+3 and stores in ECX
  - Useful for pointer arithmetic and simple calculations

- **PUSH source**: Push onto stack
  - `PUSH EAX` - Decrements ESP by 4, then stores EAX at [ESP]

- **POP destination**: Pop from stack
  - `POP EAX` - Loads value from [ESP] into EAX, then increments ESP by 4

Arithmetic and Logic

- **ADD destination, source**: Addition
  - `ADD EAX, 10` - Adds 10 to EAX

- **SUB destination, source**: Subtraction
  - `SUB ESP, 16` - Allocates 16 bytes on stack

- **XOR destination, source**: Bitwise XOR
  - `XOR EAX, EAX` - Zeros out EAX (common idiom)

- **CMP destination, source**: Compare
  - `CMP EAX, 0` - Compares EAX with 0, sets flags

Control Flow

- **JMP destination**: Unconditional jump
  - `JMP label` - Jumps to specified label

- **Conditional jumps**: Jump based on flag values
  - `JE/JZ label` - Jump if Equal/Zero
  - `JNE/JNZ label` - Jump if Not Equal/Not Zero
  - `JB/JL label` - Jump if Below/Less
  - `JA/JG label` - Jump if Above/Greater

- **CALL destination**: Call subroutine
  - `CALL function` - Pushes return address, jumps to function
- **RET**: Return from subroutine
  - `RET` - Pops return address, jumps to it

6.3.4 Status Flags

The FLAGS register contains bits that reflect the result of operations:

- **ZF** (Zero Flag): Set if the result is zero
- **SF** (Sign Flag): Set if the result is negative
- **CF** (Carry Flag): Set if there's an unsigned overflow
- **OF** (Overflow Flag): Set if there's a signed overflow

These flags are used by conditional jump instructions to determine whether to jump.

6.4 Calling Conventions

Calling conventions define how functions receive parameters and return values, how stack frames are managed, and which registers must be preserved across function calls.

6.4.1 cdecl Calling Convention

The cdecl (C Declaration) calling convention is commonly used in C programs:

- Parameters are pushed onto the stack from right to left
- The caller cleans up the stack after the function returns
- Return value is placed in EAX
- EAX, ECX, and EDX are caller-saved registers
- EBX, ESI, EDI, EBP, and ESP are callee-saved registers

Function Prologue

The standard function prologue creates a new stack frame:

```
1 push ebp          ; Save old frame pointer
2 mov ebp, esp      ; Set up new frame pointer
3 sub esp, X        ; Allocate X bytes for local variables
```

Function Epilogue

The epilogue restores the previous stack frame:

```
mov esp, ebp     ; Restore stack pointer
pop ebp          ; Restore frame pointer
ret              ; Return to caller
```

Stack Frame Layout

From high to low addresses:

- Function arguments (EBP+8, EBP+12, etc.)
- Return address (EBP+4)
- Saved EBP (EBP+0)
- Local variables (EBP-4, EBP-8, etc.)

6.4.2 System V AMD64 Calling Convention

The 64-bit calling convention for Unix-like systems:

- First six integer/pointer arguments in registers: RDI, RSI, RDX, RCX, R8, R9
- Additional arguments on the stack
- Return value in RAX
- Caller-saved registers: RAX, RCX, RDX, RSI, RDI, R8-R11
- Callee-saved registers: RBX, RBP, R12-R15
- Red zone: 128 bytes below RSP available for temporary storage

6.5 Debugging with GDB

GDB (GNU Debugger) is a powerful tool for analyzing programs during execution. It allows you to:

- Start programs with controlled environment
- Stop execution at specified conditions
- Examine program state when stopped
- Modify program state
- Track program execution

6.5.1 Basic GDB Commands

- **run**: Start the program
- **break *function/line/address***: Set a breakpoint
- **continue**: Continue execution after a breakpoint
- **step**: Execute one source line, stepping into functions
- **next**: Execute one source line, stepping over functions
- **print *expression***: Display value of expression
- **x/*format address***: Examine memory at address
- **info registers**: Display register values
- **disassemble *function***: Show assembly code
- **bt**: Show backtrace (call stack)
- **set *variable*=*value***: Modify variable value

6.5.2 Advanced GDB Features

- **Watchpoints**: Stop when a variable's value changes
- **Catchpoints**: Stop when specific events occur
- **Conditional breakpoints**: Break only when a condition is true
- **Scripting**: Automate debugging tasks with Python
- **Remote debugging**: Debug programs running on different machines
- **Core dump analysis**: Examine program state after a crash

6.5.3 GDB Extensions

Several extensions enhance GDB's functionality:

- **PEDA** (Python Exploit Development Assistance): Enhances display of data structures and memory
- **GEF** (GDB Enhanced Features): Modern interface with exploit development features
- **pwndbg**: Designed specifically for capture-the-flag (CTF) challenges

6.6 Advanced Reverse Engineering Techniques

6.6.1 Symbolic Execution

Symbolic execution is an analysis technique that explores program behavior by treating inputs as symbols rather than concrete values.

- Allows analyzing many execution paths simultaneously
- Constructs symbolic expressions representing program state
- Uses constraint solvers to determine if a path is executable
- Identifies specific inputs that lead to particular program states

How It Works

1. Initialization of input variables with symbolic values
2. Execution of the program while maintaining a symbolic state
3. At each branch, the path constraint is updated
4. The constraint solver verifies the satisfiability of paths

```
// Example program to analyze
void example(int x) {
    int y = x + 10;
    if (y > 15) {
        // Path A
        if (x * 2 == 20) {
            // Vulnerability
        }
    } else {
        // Path B
    }
}
```

Limitations

- Path explosion problem
- Difficulty handling external function calls
- Complexity in modeling complex memory operations
- Limitations of constraint solvers for certain classes of constraints

6.6.2 Angr Framework

Angr is a Python binary analysis framework that combines static analysis, symbolic execution, and other techniques.

Key Features

- **Multi-architecture**: supports x86, x86-64, ARM, MIPS, and others
- **Intermediate representation**: uses VEX IR to abstract from specific ISAs
- **Static analysis**: CFG, value analysis, program slicing
- **Symbolic execution**: path exploration and input generation
- **Concrete execution**: to handle sections difficult to analyze symbolically

Typical Usage

```
import angr

# Load the binary
proj = angr.Project('target_binary')

# Create an initial state
state = proj.factory.entry_state()

# Create a simulator
simgr = proj.factory.simulation_manager(state)

# Explore until reaching a target
simgr.explore(find=0x400abcd)  # Address of interest

# Analyze results
if simgr.found:
    solution = simgr.found[0].posix.dumps(0)  # Input
    that reaches the target
    print(solution)
```

Applications

- **Vulnerability research**: Automated identification of bugs
- **Exploit development**: Payload generation to exploit vulnerabilities
- **Crackme and CTF**: Automated solving of challenges

- **Malware analysis**: Extraction of behaviors and configurations
- **Protocol reverse engineering**: Deduction of proprietary protocol formats

Chapter 7

Hardware Security

7.1 Introduction to Hardware Security

Hardware security focuses on protecting computer systems from vulnerabilities at the hardware level. This includes protecting against physical attacks, side-channel attacks, and hardware-based vulnerabilities.

7.2 Meltdown Attack

7.2.1 What is Meltdown?

Meltdown (CVE-2017-5754) is a hardware vulnerability that affects most Intel processors manufactured since 1995. It allows malicious programs to read from protected memory areas, potentially exposing sensitive data such as passwords and encryption keys.

7.2.2 Technical Background

Meltdown exploits a feature called speculative execution:

- Modern processors execute instructions speculatively to improve performance
- When a program accesses memory, the processor checks permissions
- However, the processor may speculatively execute instructions before completing the permission check
- Although the results of unauthorized memory accesses are eventually discarded, they can temporarily affect cache state
- This cache state can be detected using side-channel techniques

7.2.3 Meltdown Attack Mechanism

The Meltdown attack follows these steps:

1. The attacker sets up a "probe array" in memory
2. The attacker attempts to read from a protected memory location
3. The processor speculatively executes this read before determining it's unauthorized
4. The value read is used to access an index in the probe array, affecting the cache
5. The speculative execution is eventually rolled back when the processor realizes the access was unauthorized
6. However, the cache state changes remain
7. The attacker measures access times to each element in the probe array
8. A faster access time for a particular index indicates that value was in the protected memory

7.2.4 Impact of Meltdown

Meltdown allows:

- Reading of kernel memory from user space
- Bypassing of memory isolation between processes
- Extracting sensitive data, including passwords and encryption keys
- Compromising virtualization boundaries (in some cases)

7.2.5 Countermeasures

- **Kernel Page Table Isolation (KPTI)**: Separates kernel and user address spaces completely
- **Processor microcode updates**: Mitigates some aspects of the vulnerability
- **New processor designs**: Intel's newer processors include hardware mitigations
- **Special handling of sensitive data**: Storing sensitive data in protected enclaves

7.3 Spectre Attack

7.3.1 What is Spectre?

Spectre (CVE-2017-5753 and CVE-2017-5715) is a class of hardware vulnerabilities that affects nearly all modern processors, including those from Intel, AMD, and ARM. It allows attackers to trick programs into revealing their data.

7.3.2 Technical Background

Like Meltdown, Spectre exploits speculative execution. However, while Meltdown focuses on privilege boundaries, Spectre targets different security mechanisms:

- **Branch prediction**: Processors predict which branch of code will execute next
- **Out-of-order execution**: Instructions are executed in an order different from the program's apparent order

7.3.3 Spectre Variants

Variant 1: Bounds Check Bypass

This variant exploits conditional branch prediction:

1. Attackers train the branch predictor to anticipate a specific branch direction
2. When the actual code executes, the processor speculatively follows the predicted path
3. If the prediction was incorrect, the results are eventually discarded
4. However, as with Meltdown, cache state changes persist and can be detected

Variant 2: Branch Target Injection

This variant manipulates indirect branch prediction:

1. Attackers train the branch target buffer to predict specific jump targets
2. When the vulnerable code executes an indirect jump, it speculatively jumps to the attacker-controlled location
3. This allows the attacker to execute arbitrary speculative code
4. Again, cache state changes can reveal sensitive information

7.3.4 Impact of Spectre

Spectre allows:

- Reading memory from other processes
- Extracting data from within the same process (e.g., in sandboxed JavaScript)
- Bypassing software-based security controls
- Potentially attacking across virtualization boundaries

7.3.5 Countermeasures

- **Microcode updates**: Introduce new features like Indirect Branch Prediction Barriers
- **Retpoline**: A software mitigation that replaces vulnerable indirect jumps
- **Site isolation**: Browsers use separate processes for different websites
- **Speculative load hardening**: Prevents speculative loads from leaking information
- **New processor designs**: Include hardware mitigations for known variants

7.4 Comparison of Meltdown and Spectre

7.5 Broader Implications for Hardware Security

Meltdown and Spectre demonstrated that hardware optimizations can have significant security implications:

- Highlights the tension between performance and security
- Shows that hardware vulnerabilities can persist for decades before discovery
- Emphasizes the need for hardware-software co-design for security
- Encourages formal verification of processor designs
- Prompts reconsideration of threat models in computing systems

Aspect	**Meltdown**	**Spectre**
Affected processors	Primarily Intel	Almost all modern processors (Intel, AMD, ARM)
Exploitation	Easier to exploit, more reliable	More difficult to exploit, context-dependent
Target	Primarily targets privilege boundaries (user/kernel)	Can target any security boundary within a process or between processes
Mitigation	KPTI provides effective mitigation	More complex to mitigate, requires multiple strategies
Performance impact	Significant performance impact (5-30%)	Variable impact depending on mitigation strategy
Long-term solution	Requires hardware redesign	Requires hardware redesign

Table 7.1: Comparison of Meltdown and Spectre

Chapter 8

Conclusion

8.1 Summary of Key Concepts

This guide has covered a wide range of topics in ethical hacking and cybersecurity, including:

- Network security fundamentals and attacks
- Web application vulnerabilities
- Software exploitation techniques
- Blockchain security
- Reverse engineering methods
- Hardware security vulnerabilities

8.2 The Ethical Hacker's Mindset

Ethical hacking requires more than technical knowledge—it demands a specific mindset:

- Think creatively about potential attack vectors
- Understand both offensive and defensive perspectives
- Stay updated on emerging threats and vulnerabilities
- Consider the ethical implications of security research
- Respect legal boundaries and obtain proper authorization
- Document findings thoroughly and communicate them responsibly
- Contribute to improving overall security posture

8.3 Future Trends in Cybersecurity

As technology evolves, so do cybersecurity challenges and ethical hacking practices:

- Artificial intelligence and machine learning in both attacks and defenses
- Quantum computing threats to existing cryptographic systems
- Supply chain security becoming increasingly critical
- Internet of Things (IoT) security challenges
- Cloud security requiring specialized approaches
- Privacy-preserving technologies gaining importance
- Zero-trust architecture becoming the standard model

8.4 Resources for Further Learning

8.4.1 Books

- "The Shellcoder's Handbook: Discovering and Exploiting Security Holes"
- "Hacking: The Art of Exploitation"
- "Practical Malware Analysis"
- "The Web Application Hacker's Handbook"
- "Reversing: Secrets of Reverse Engineering"

8.4.2 Online Resources

- OWASP (Open Web Application Security Project)
- Exploit Database
- CTF (Capture The Flag) competitions
- HackTheBox and TryHackMe platforms
- National Vulnerability Database (NVD)

8.4.3 Tools

- Network analysis: Wireshark, tcpdump
- Vulnerability scanning: Nmap, OpenVAS
- Web application testing: Burp Suite, OWASP ZAP
- Reverse engineering: Ghidra, IDA Pro, Radare2
- Exploitation: Metasploit Framework
- Digital forensics: Autopsy, Volatility

Appendix A

Glossary of Terms

ASLR (Address Space Layout Randomization)
A security technique that randomizes the address space positions of key program components, making it harder for attackers to predict target addresses.

Buffer Overflow
A condition where a program writes data beyond the boundaries of allocated memory, potentially allowing attackers to manipulate program execution.

CSRF (Cross-Site Request Forgery)
An attack that forces authenticated users to execute unwanted actions on websites where they're logged in.

DLT (Distributed Ledger Technology)
A decentralized system for recording transactions across multiple computers.

ELF (Executable and Linkable Format)
The standard binary format for executables and libraries in Unix-like systems.

Format String Vulnerability
A security flaw where user input is incorrectly used as a format string in functions like printf().

NIC (Network Interface Card)
Hardware component that connects a computer to a network.

ROP (Return-Oriented Programming)
An exploitation technique that chains together existing code fragments to perform malicious operations.

Shellcode
: Small pieces of code used as payloads in exploiting software vulnerabilities.

SQL Injection
: An attack technique that inserts malicious SQL code into database queries.

TCP Three-Way Handshake
: The process used to establish a TCP connection between two endpoints.

XSS (Cross-Site Scripting)
: A vulnerability that allows attackers to inject client-side scripts into webpages viewed by others.