

# UNIT -01

## Introduction of system security

---

### **1. Control Hijacking Attack**

#### **Definition:**

Control hijacking attack is a type of cyber attack where a hacker takes control of a program's normal execution flow. The attacker exploits vulnerabilities in software to execute malicious code, often leading to system compromise.

#### **How it Works:**

- Hackers target software bugs, such as buffer overflows.
- They inject malicious code into the program's memory.
- When the program runs, it unknowingly executes the attacker's code instead of normal instructions.
- This can give the attacker unauthorized access, system control, or ability to steal data.

#### **Types of Control Hijacking Attacks:**

1. **Buffer Overflow Attack:** Overflowing a program's memory to overwrite execution control.
2. **Return-to-libc Attack:** Redirecting program execution to existing library functions.
3. **Format String Attack:** Exploiting improper use of string formatting in code.

#### **Prevention and Control Measures:**

- **Input Validation:** Always check input data length and type.
- **Safe Programming Practices:** Avoid unsafe functions and use secure coding.
- **Memory Protection:** Use stack canaries, non-executable stacks, and address space layout randomization (ASLR).
- **Regular Updates:** Keep software patched to fix known vulnerabilities.

#### **Conclusion:**

Control hijacking is dangerous because it allows attackers to take over programs and systems. Using secure coding, memory protections, and updates can significantly reduce these attacks.

---

### **2. Buffer Overflow**

#### **Definition:**

A buffer overflow occurs when a program writes more data into a buffer (a temporary storage area in memory) than it can hold. This extra data can overwrite adjacent memory, causing unexpected behavior or allowing attackers to take control.

### How it Happens:

- Programs reserve memory space (buffer) for input or data.
- If input exceeds the buffer size, it overflows into nearby memory.
- This can corrupt data, crash the program, or let attackers execute malicious code.

### Example:

A program expects 10 characters, but the user enters 50. Extra 40 characters overflow memory.

### Consequences:

- Program crash
- Data corruption
- Security breach (attacker can run malicious code)

### Defences:

1. **Input Validation:** Check length and type of all inputs.
2. **Safe Functions:** Use secure alternatives (e.g., `strncpy()` instead of `strcpy()`).
3. **Stack Canaries:** Special values placed on the stack to detect overflow.
4. **Non-Executable Stack/Heap:** Prevents execution of injected code.
5. **ASLR (Address Space Layout Randomization):** Randomizes memory addresses to make exploitation harder.
6. **Code Auditing and Testing:** Detects vulnerabilities during development.

### Conclusion:

Buffer overflow is a common attack method. Proper input validation and safe coding practices can prevent it.

---

## 3. Integer Overflow

### Definition:

Integer overflow happens when a calculation produces a number larger than the storage limit of an integer variable. The extra value “wraps around,” causing incorrect results.

- **How it Occurs:** Each integer type has a maximum and minimum value.
- Adding, multiplying, or incrementing beyond these limits causes overflow.
- Example: For an 8-bit unsigned integer, max value is 255. Adding 1 results in 0.

### Consequences:

- Incorrect calculations
- Unexpected program behavior
- Potential security vulnerabilities

### Prevention:

- Use larger data types
- Perform range checks
- Use safe arithmetic libraries

Integer overflow can cause errors and security issues if not handled properly.

---

## **4. Bypassing Browser Memory Protection**

### **Definition:**

Browser memory protection is designed to prevent attackers from executing malicious code by safeguarding memory areas. Mechanisms include **ASLR (Address Space Layout Randomization)**, **DEP (Data Execution Prevention)**, and **stack canaries**.

### **How Bypassing Happens:**

- Hackers exploit software bugs like buffer overflows or use-after-free errors.
- They manipulate memory to execute arbitrary code despite protections.
- Techniques like **Return-Oriented Programming (ROP)** chain existing safe code snippets to bypass protections.
- **Heap spraying** is used to fill memory with attacker-controlled code to increase the chance of execution.

### **Consequences:**

- Hijacked program control
- Data theft or corruption
- System compromise

### **Prevention:**

- Keep browsers and plugins updated
- Enable built-in security features
- Use sandboxed browsing
- Avoid untrusted websites and downloads

### **Conclusion:**

Bypassing browser memory protection is a critical attack method. Proper updates, security configurations, and safe browsing habits can reduce risks significantly.

---

## **Sandboxing and Isolation**

### **Definition:**

Sandboxing and isolation are security techniques used to protect a system from malicious programs or untrusted code.

## **Sandboxing:**

- Runs a program in a **restricted environment**.
- Limits access to system resources like files, network, and memory.
- Even if the program is malicious, it cannot harm the rest of the system.
- Example: Web browsers running JavaScript in a sandbox.

## **Isolation:**

- Separates applications or processes from each other.
- Prevents one program from affecting another or the operating system.
- Example: Virtual machines isolate an entire OS from the host system.

## **Types of Sandboxing:**

1. **Application Sandbox:** Runs a single program with restricted access to system resources.
  - Example: Web browsers isolating JavaScript execution.
2. **Virtual Machine (VM) Sandbox:** Runs an entire OS in a virtual environment, fully isolated from the host.
  - Example: VirtualBox, VMware.
3. **Container-Based Sandbox:** Uses lightweight containers to isolate applications while sharing the OS kernel.
  - Example: Docker containers.
4. **Cloud Sandboxing:** Runs code or applications in a remote cloud environment to prevent local system compromise.

## **Types of Isolation:**

1. **Process Isolation:** Each application runs in a separate process with its own memory space.
2. **User Account Isolation:** Different user accounts prevent access to others' data and programs.
3. **Network Isolation:** Restricts network communication for certain programs to limit exposure.
4. **Hardware-Based Isolation:** Uses CPU features like Intel VT-x or AMD-V to separate environments.

## **Benefits:**

- Protects sensitive data from attacks.
- Reduces damage from malware or crashes.
- Enables safe testing of untrusted applications.

## **Conclusion:**

Both sandboxing and isolation improve system security by containing programs within controlled environments, preventing them from accessing critical resources or interfering with other applications.

## **5. Security Vulnerabilities Detection Tools**

### **Definition:**

These are tools used to identify weaknesses or flaws in software, networks, or systems that could be exploited by attackers.

### **Types and Techniques:**

#### **1. Static Analysis Tools:**

- Analyze source code without executing it.
- Detect coding errors, buffer overflows, and insecure functions.
- Example: **SonarQube, Coverity**

#### **2. Dynamic Analysis Tools:**

- Monitor software during execution to find runtime vulnerabilities.
- Detect memory leaks, crashes, and improper access.
- Example: **Valgrind, Purify**

#### **3. Penetration Testing Tools:**

- Simulate attacks to find system vulnerabilities.
- Example: **Metasploit, Burp Suite**

#### **4. Fuzzing Tools:**

- Provide random or malformed input to programs to detect crashes or unexpected behavior.
- Example: **AFL (American Fuzzy Lop), Peach Fuzzer**

### **Conclusion:**

Using these tools during development and testing helps identify and fix security flaws early, making software and systems more secure.

---

## **6. Privileges and Access Control**

### **Definition:**

Privileges are the rights or permissions assigned to a user or process, defining what actions they can perform. Access control enforces these privileges to restrict unauthorized access to system resources.

### **Types of Access Control:**

#### **1. Discretionary Access Control (DAC):**

- The **resource owner** decides who can access it.
- Flexible but less secure because users can grant access to others.
- Example: File permissions in Windows or Linux.

## 2. **Mandatory Access Control (MAC):**

- **System-enforced rules** control access based on security labels (e.g., classified, secret).
- Users cannot change permissions.
- Example: Military or government systems.

## 3. **Role-Based Access Control (RBAC):**

- Access is granted based on **user roles** rather than individual identity.
- Easier to manage in large organizations.
- Example: Admin, Manager, Employee roles in enterprise software.

## 4. **Attribute-Based Access Control (ABAC):**

- Access is determined by **attributes** like user, resource, environment, or action.
- More dynamic and fine-grained control.
- Example: Allow access only during office hours from corporate devices.

### **Importance:**

- Protects sensitive data
- Maintains confidentiality, integrity, and availability
- Prevents unauthorized actions

### **Conclusion:**

Choosing the right type of access control strengthens system security and reduces risks of data breaches.

---

## **Operating System Security**

### **Definition:**

Operating System (OS) security protects data, programs, and resources from unauthorized access, attacks, or misuse.

### **Key Features:**

- **Authentication:** Verifies user identity through passwords or biometrics.
- **Access Control:** Restricts resources using DAC, MAC, or RBAC.
- **Process & Memory Protection:** Prevents one process from affecting others.
- **Audit & Logging:** Monitors activities for security and troubleshooting.
- **Encryption & Patching:** Protects data and fixes vulnerabilities.

**Conclusion:**

Strong OS security ensures system integrity, confidentiality, and availability, preventing unauthorized access, malware attacks, and data breaches.

---

## **Exploitation Techniques**

**Definition:**

Exploitation techniques are methods used by attackers to take advantage of software, system, or network vulnerabilities to gain unauthorized access or control.

**Common Techniques:**

1. **Buffer Overflow:** Overwriting memory to execute malicious code.
2. **Code Injection:** Injecting harmful code into applications (e.g., SQL injection, cross-site scripting).
3. **Privilege Escalation:** Gaining higher-level access than permitted.
4. **Return-Oriented Programming (ROP):** Using existing program code snippets to bypass protections.
5. **Social Engineering:** Tricking users to reveal credentials or execute malware.

**Consequences:**

- Unauthorized access
- Data theft or corruption
- System compromise

**Prevention:**

- Regular software updates
- Input validation and secure coding
- Strong access control and monitoring

**Conclusion:**

Exploitation techniques are dangerous but preventable through secure coding, system hardening, and user awareness.

---

## **Fuzzing**

**Definition:**

Fuzzing is a software testing technique that provides random, unexpected, or malformed inputs to programs to detect bugs, crashes, or security vulnerabilities.

### **How it Works:**

- The fuzzer sends large amounts of random data to the application.
- Monitors program behavior for crashes, memory leaks, or abnormal responses.
- Helps discover security weaknesses before attackers exploit them.

### **Types of Fuzzing:**

1. **Black-Box Fuzzing:** No knowledge of internal code; tests only inputs and outputs.
2. **White-Box Fuzzing:** Uses source code or internal program knowledge to generate test cases.
3. **Grey-Box Fuzzing:** Partial knowledge of the program; combines black-box and white-box methods.
4. **Mutation-Based Fuzzing:** Modifies existing valid inputs to create test cases.
5. **Generation-Based Fuzzing:** Generates inputs from scratch based on protocol or input format.

### **Conclusion:**

Fuzzing is an effective way to detect software vulnerabilities early, improving reliability and security.

# UNIT -02

## Software Security

---

### **Privileged Programs (Set-UID Programs) and Vulnerabilities**

#### **Definition:**

Set-UID programs are executable files in Unix/Linux that run with the **privileges of the file owner**, often the root user, instead of the user executing them.

#### **Vulnerabilities:**

- **Buffer Overflow:** Attackers can overwrite memory to execute malicious code with elevated privileges.
- **Race Conditions:** Timing issues can allow attackers to access or modify files unexpectedly.
- **Improper Input Validation:** Malicious input can exploit the program to gain higher privileges.

### **Privilege Separation**

#### **Definition:**

Privilege separation is a security technique where a program is divided into parts with **different privilege levels**.

#### **How it Works:**

- Only the component that needs elevated privileges runs with high-level rights.
- The rest of the program runs with normal user privileges.

#### **Benefits:**

- Limits the impact of a vulnerability.
- Prevents attackers from gaining full system access if one part of the program is exploited.
- Encourages safer design of programs handling sensitive operations.

#### **Conclusion:**

Set-UID programs are essential but risky. Applying **privilege separation** reduces the risk of system compromise by containing potential attacks to only the part requiring higher privileges.

## Return-to-libc Attack

### Definition:

Return-to-libc is an advanced exploitation technique used to bypass **non-executable stack protections**. Instead of injecting malicious code, the attacker redirects a program's execution to existing **library functions** (like `system()` in `libc`).

### How It Works:

1. Exploits a **buffer overflow** to overwrite the function's return address on the stack.
2. Instead of jumping to attacker-injected code, the program jumps to a standard library function.
3. Arguments (e.g., `/bin/sh`) are passed via the stack to execute commands.

### Consequences:

- Allows attackers to execute shell commands.
- Can escalate privileges if targeting a privileged program.
- Harder to detect since no new code is injected.

### Prevention:

- Use **stack canaries** to detect overflows.
- Enable **ASLR (Address Space Layout Randomization)** to randomize library addresses.
- Apply **non-executable memory protections**.
- Write secure code with proper **input validation**.

### Conclusion:

Return-to-libc attacks are a sophisticated method to exploit vulnerable programs without injecting code, emphasizing the need for memory protections and secure programming practices.

---

## Race Condition Vulnerability

### Definition:

A race condition happens when a program's result depends on the order or timing of two or more processes. If the program does not handle this properly, attackers can exploit it.

### How It Can Be Exploited:

Attackers take advantage of the time gap between checking a resource and using it. They act faster than the program expects to access or change something they shouldn't.

### Example:

A program creates a temporary file and then opens it. An attacker quickly replaces the file

with a link to an important system file (like `/etc/passwd`). The program writes to that system file, allowing the attacker to change it.

### Prevention:

- Use **locks** or **mutexes** to control access.
- Don't use predictable file names.
- Always check resources again before using them.

### Conclusion:

Race conditions are dangerous but can be avoided with careful programming and proper synchronization.

---

## Dirty COW Attack

### Definition:

Dirty COW (Copy-On-Write) is a serious Linux vulnerability that allows a normal user to **modify read-only files** and gain **root (administrator) privileges** without proper authorization.

### Factors Leading to Dirty COW:

1. **Race Condition in Memory Management:** The bug exploits a timing flaw in the copy-on-write mechanism of the Linux kernel. This allows changes to memory before the system can make a proper copy.
2. **Unprotected Kernel Memory:** Certain areas of memory that should be read-only can be changed if the vulnerability is exploited.
3. **User Privileges Misuse:** Any unprivileged user can use this bug to escalate their privileges and take control of the system.

### How the Attack Works:

- The attacker repeatedly writes to a memory mapping of a read-only file.
- Due to the race condition, the system copies the file incorrectly, letting the attacker overwrite data.
- This can be used to modify sensitive files like `/etc/passwd` to gain root access.

### Preventive Measures:

- Regularly **update the Linux kernel** to apply patches.
- Use **privilege separation** and avoid running untrusted code as root.
- Enable security modules like **SELinux** or **AppArmor**.
- Limit user access and monitor system changes.

**Conclusion:**

Dirty COW is a critical vulnerability caused by a memory race condition. Proper updates, privilege management, and security tools can prevent exploitation and protect Linux systems.

---

## **Format String Vulnerability and Attack**

**Definition:**

Format string vulnerability occurs when a program **improperly uses user input as a format string** in functions like `printf`. This can allow attackers to read or modify memory.

**Factors Leading to Vulnerability:**

- **Improper Input Handling:** User input is directly used in format functions without validation.
- **Unsafe Functions:** Using functions like `printf` or `sprintf` without care.
- **Lack of Memory Protection:** Can lead to arbitrary memory read/write.

**How the Attack Works:**

- The attacker provides input with format specifiers like `%x` or `%n`.
- `%x` can **read memory values**, `%n` can **overwrite memory**.
- This can be used to manipulate program behavior or execute malicious code.

**Preventive Measures:**

- Always **validate and sanitize user input**.
- Avoid passing user input directly to format functions.
- Use safer alternatives like `snprintf` and proper coding practices.

**Conclusion:**

Format string vulnerabilities are dangerous but preventable through careful input validation, safe coding, and memory protection.

---

## **Shellshock Attack**

**Definition:**

Shellshock is a **vulnerability in the Bash shell** that allows attackers to execute arbitrary commands using environment variables.

**Factors Leading to Vulnerability:**

- **Bash Processing of Environment Variables:** Untrusted input can be executed.
- **Remote Access Exposure:** Attackers can send malicious variables via network services.
- **Lack of Patching:** Older Bash versions are vulnerable.

#### **How the Attack Works:**

- Attacker sets a malicious environment variable.
- Bash executes the commands automatically when processing the variable.
- This can lead to remote code execution and server compromise.

#### **Preventive Measures:**

- Update Bash to patched versions.
- Limit exposure of services to untrusted networks.
- Use secure configurations and monitoring.

#### **Conclusion:**

Shellshock allows remote command execution; proper patching and secure configurations prevent exploitation.

---

## **Heartbleed Attack**

#### **Definition:**

Heartbleed is a vulnerability in **OpenSSL's heartbeat feature** that allows attackers to read memory from servers.

#### **Factors Leading to Vulnerability:**

- **Improper Input Validation:** Malformed heartbeat requests are accepted.
- **Sensitive Memory Exposure:** Server memory may contain passwords, keys, or user data.
- **Unpatched OpenSSL Versions:** Older versions are vulnerable.

#### **How the Attack Works:**

- Attacker sends a specially crafted heartbeat request.
- The server responds with more data than it should, exposing sensitive memory.
- This can leak private keys, passwords, and user information.

#### **Preventive Measures:**

- Update OpenSSL to fixed versions.
- Disable vulnerable services and protocols.
- Use proper server configurations and monitoring.

**Conclusion:**

Heartbleed allows attackers to steal sensitive data remotely. Timely updates and secure configurations prevent exploitation.

---

**Interactivity, Annotation, and Arrangement****Definition:**

These are software testing and development techniques used to improve **program analysis, security, and maintainability**.

**Factors Leading to Issues (If Ignored):**

- **Interactivity:** Lack of proper monitoring can hide user or system input errors.
- **Annotation:** Missing or unclear metadata and comments can make vulnerabilities harder to detect.
- **Arrangement:** Poor organization of code and components can lead to bugs and security flaws.

**How They Work / Usage:**

- **Interactivity:** Observes user and system actions to detect abnormal behavior or vulnerabilities.
- **Annotation:** Developers add metadata, comments, or markers in code to track potential security issues.
- **Arrangement:** Properly structures program modules, functions, and resources to reduce errors and security risks.

**Benefits / Preventive Measures:**

- Improves detection of errors and vulnerabilities.
- Makes code easier to understand, maintain, and audit.
- Reduces security risks by organizing code logically and clearly.
- Helps testers and security analysts follow program behavior systematically.

**Conclusion:**

Interactivity, annotation, and arrangement are essential for **secure, maintainable, and reliable software development**. Ignoring them can lead to hidden bugs and vulnerabilities, while proper use enhances program security and analysis.