Buffer Overflow

Definition

A buffer overflow occurs when a program writes more data into a buffer (a fixed-size memory block) than it can hold. This can overwrite adjacent memory locations, leading to unpredictable behavior, crashes, and even security vulnerabilities.

Example:

```
char sample[10];

int i;

for (i=0; i<=9; i++) sample[i] = ‘A’;

sample[10] = ‘B’;
```

Harm from Buffer Overflows

- **Overwrite:**
  - Another piece of your program’s data
  - An instruction in your program
  - Data or code belonging to another program
  - Data or code belonging to the operating system
- **Security Impact:**
  - Overwriting a program’s instructions gives attackers that program’s execution privileges.
  - Overwriting operating system instructions gives attackers the operating system’s execution privileges.
- **DOS(Denial of Service) :**

  Simply providing random input can cause the program to crash. This makes it unable to function and provide the service it was meant to give. Thus it leads to a denial of service.

  A more sophisticated attack can modify the return value and control the program execution to run statements of the attackers choosing, thus giving him access to the code execution privileges of the program.

Common Causes of Buffer Overflows

Buffer overflows often occur due to improper memory handling in languages like C and C++. The most common causes include:

- **Lack of Bounds Checking:** Writing data without verifying its size can lead to buffer overflows.
- **Unsafe String Functions:** Functions that do not enforce size limits can overwrite adjacent memory.
    - **Vulnerable C Functions:**
        - `gets(buffer)` – Reads input without size restrictions.
        - `strcpy(dest, src)` – Copies a string without checking destination size.
        - `strcat(dest, src)` – Appends a string without checking size.
        - `sprintf(dest, format, args...)` – Formats output but does not check bounds.
        - `scanf("%s", buffer)` – Reads user input without bounds checking.

`gets(char *str)`	read line from standard input into str
`sprintf(char *str, char *format, ...)`	create str according to supplied format and variables
`strcat(char *dest, char *src)`	append contents of string src to string dest
`strcpy(char *dest, char *src)`	copy contents of string src to string dest
`vsprintf(char *str, char *fmt, va_list ap)`	create str according to supplied format and variables

- **Stack-Based Overflow:** Writing past a function's local variable space can overwrite the return address.

Buffer Overflow Defenses

Buffer overflow defenses can be broadly classified into two approaches:

- **Compile-time Defenses:** Harden programs during compilation to prevent buffer overflow attacks in new programs.
- **Run-time Defenses:** Detect and abort attacks in already deployed, existing programs.

Despite known defensive techniques, the vast base of vulnerable software limits their widespread deployment. Therefore, **run-time defenses** are crucial as they can be applied to existing software through operating system updates and security patches.

Compile-Time Defenses

1. Programming Language Selection

- Using modern high-level languages (e.g., Python, Java, Rust) prevents buffer overflows.
- Compilers enforce range checking and restrict unsafe memory operations.

Disadvantages:

- Additional runtime overhead due to checks.
- Less control over low-level hardware, limiting their use in device drivers and system programming.

2. Safe Coding Techniques

- The C programming language prioritizes performance over type safety, making it vulnerable.
- Secure coding practices require **manual code reviews** to identify and eliminate unsafe coding patterns.
- Example: The **OpenBSD project** audited codebases, leading to one of the safest operating systems in use.

3. Language Extensions & Safe Libraries

- Handling **dynamically allocated memory** is challenging since size information isn't always available at compile time.
- Requires recompilation and compatibility with third-party applications.
- **Example: Libsafe** – A dynamic library that replaces unsafe C standard library functions with safer alternatives.

4. Stack Protection Mechanisms

- **Stack Canaries:**
    - A random value placed before the return address; if modified, the program detects corruption and aborts execution.
    - Must be unpredictable and unique per system to prevent attackers from guessing its value.
- **GCC Extensions:**
    - **StackShield** and **Return Address Defender (RAD)** add function entry/exit code to detect stack corruption.
    - Function entry saves a copy of the return address in a protected memory region.

    - Function exit compares the return address before execution continues; if altered, the program terminates.

Run-Time Defenses

1. Executable Address Space Protection

- Uses virtual memory features to mark some memory regions as **non-executable** (e.g., **DEP – Data Execution Prevention**).
- Prevents execution of injected shellcode from buffer overflows.
- Historically present in **SPARC/Solaris**, now adopted in **x86 Linux, Unix, and Windows**.
- Challenge: Some programs require an **executable stack**, needing special exceptions.

2. Address Space Layout Randomization (ASLR)

- **Randomizes memory locations** for key data structures such as:
    - **Stack, heap, global data, and shared libraries.**
    - Each process gets a different memory layout, making return-to-libc and other exploits harder.
- **Modern OS implementations** ensure attackers cannot predict target memory locations.

3. Guard Pages

- Inserts **protected memory pages** between critical regions of memory.
- Marked as **illegal addresses** in the Memory Management Unit (MMU).
- Any access triggers an **automatic process termination**.
- Additional defenses: Placing **guard pages between stack frames and heap buffers** to mitigate overflows.

Incomplete Mediation

Incomplete mediation occurs when a system or application fails to properly validate, filter, or enforce access controls on user inputs before processing them. This vulnerability can allow attackers to bypass security restrictions, execute unauthorized actions, or manipulate data. It is often found in scenarios where input is expected to meet certain constraints but is not thoroughly checked before being used.

Example of Incomplete Mediation

An example is **parameter tampering in web forms**. Consider an e-commerce website where users select products and their prices are stored in hidden form fields:

```
<input type="hidden" name="price" value="100">
```

If the backend does not validate the price before processing the transaction, an attacker could manipulate the form input to change the price (e.g., modifying the hidden field to `$1` instead of `$100`). Without proper validation, the attacker could purchase products at an incorrect price.

Prevention Techniques

To prevent incomplete mediation vulnerabilities, developers should follow these best practices:

1. **Perform Input Validation at the Server-Side:**
    - Validate all user input against strict rules (e.g., type, length, format, and expected values).
    - Use whitelisting (defining acceptable values) rather than blacklisting (blocking specific values).
2. **Enforce Access Controls at Every Level:**
    - Implement **role-based access control (RBAC)** to ensure that users can only access permitted resources.
    - Verify permissions on every request, not just at login.
3. **Sanitize and Encode User Inputs:**
    - Use secure libraries to sanitize inputs and prevent injection attacks.
    - Encode output to prevent cross-site scripting (XSS) and injection attacks.

Time-of-Check to Time-of-Use (TOCTTOU) Flaw

The **Time-of-Check to Time-of-Use (TOCTTOU)** flaw occurs when there is a time gap between checking access permissions and executing an action. During this delay, an attacker can manipulate data, leading to unauthorized actions.

How TOCTTOU Works:

- A **mediator** checks access rights before allowing an operation.
- If the user alters the data (e.g., changing a file name) between the check and execution, the mediator might unknowingly approve an unintended action.
- This security flaw can compromise **confidentiality** or **integrity** by allowing unauthorized modifications.

Example:

- A system verifies that a user has permission to modify **File A**.
- Before execution, the user swaps **File A** with **File B** in the instruction, leading to unauthorized modifications or deletions.

Countermeasures:

1. **Control Request Data** – Ensure the system maintains control over request parameters until execution is complete.
2. **Enforce Serial Integrity** – Prevent any modification during validation.
3. **Copy & Validate Data** – Move data out of the user's control before validation.
4. **Seal Data** – Use cryptographic methods to detect unauthorized modifications.

By implementing these protections, systems can prevent TOCTTOU exploits and ensure secure access control.

Undocumented Access Points (Backdoors)

Undocumented access points, also known as **backdoors** or **trapdoors**, are secret entryways left in software by developers, often for debugging or maintenance. These entry points allow access to the system **without proper authentication or authorization**.

Causes of Undocumented Access Points:

- Developers create hidden **execution modes** during testing but forget to remove them.
- Some programmers **intentionally leave backdoors** for future maintenance, believing they will remain unnoticed.
- However, **attackers can discover and exploit** these access points, leading to security breaches.

Security Risks:

- **Unauthorized Access** – Attackers can bypass authentication and gain control over a system.
- **Privilege Escalation** – Backdoors may grant administrative privileges, leading to full system compromise.
- **Data Theft & System Manipulation** – Sensitive data can be stolen or modified without detection.

Preventing Undocumented Access Points:

1. **Rigorous Code Reviews** – Conduct thorough security audits to detect and eliminate hidden entry points.
2. **Strict Development Policies** – Enforce secure coding practices to prevent backdoors from being added.
3. **Automated Security Scans** – Use static and dynamic analysis tools to identify suspicious code.
4. **Access Control & Monitoring** – Implement logging and access controls to detect unauthorized system access.

Off-by-One Error

An **off-by-one error** occurs when a loop or array index is miscalculated, leading to **out-of-bounds memory access** or incorrect logic.

Example: A loop is written incorrectly:

```
for (i = 0; i <= n; i++)  // Should be i < n to avoid exceeding bounds
```

- This might cause the loop to iterate **one extra time**, potentially accessing invalid memory.

Another common mistake is misunderstanding array sizes:

```
char buffer[10];
buffer[10] = 'A';  // Off-by-one error, valid indices are 0-9
```

- This can **overwrite adjacent memory** and lead to unpredictable behavior.
- **Security Impact:** Can lead to **buffer overflows**, memory corruption, or program crashes.
- **Prevention:** Proper boundary checking, static analysis tools, and secure coding practices.

Integer Overflow

An **integer overflow** occurs when a calculation exceeds the storage limit of a variable, leading to unexpected values.

Example:

```
int x = 2147483647; // Max for a 32-bit signed int
x = x + 1;  // Overflow: wraps to -2147483648
```

- This can **disrupt program logic** or be exploited in attacks such as **buffer overflow exploits**.
- **Security Impact:** Attackers may exploit integer overflows to bypass security checks, causing **denial of service (DoS) or privilege escalation**.
- **Prevention:** Use compiler checks, **bounds validation**, and safe arithmetic functions to detect overflow conditions.

Unterminated Null-Terminated Strings

Many programming languages use **null-terminated strings**, meaning a **null byte (0x00)** marks the end of a string. If this byte is accidentally removed or overwritten, the program may **continue reading memory until it finds another null byte**, leading to data leakage or crashes.

Example:

```
char str[5] = {'H', 'e', 'l', 'l', 'o'}; // No null terminator
printf("%s", str); // Undefined behavior: continues reading past
allocated memory
```

- **Security Impact:** May cause **buffer overflows**, leaking **sensitive memory contents** or allowing arbitrary code execution.
- **Prevention:** Use **safer string functions**, **explicit length checks**, and **memory-safe languages** where possible.

Parameter Length, Type, and Number Issues

Functions may receive incorrect or unexpected inputs, causing errors such as **buffer overflows, format string vulnerabilities, or type mismatches**.

- **Common Issues:**
  - Passing **too many** or **too few** parameters.
  - Providing a **string longer than expected**, leading to overflow.
  - Supplying an **unexpected data type**, causing crashes.

```
void process_input(char *input) {
    char buffer[10];
    strcpy(buffer, input); // No length check, can overflow
}
```

- **Security Impact:** Attackers may exploit these issues to **execute arbitrary code or crash a system**.
- **Prevention:** Perform **strict input validation**, enforce **type safety**, and use **secure function alternatives**.

Unsafe Utility Functions

Many **standard library functions** in languages like **C** do not perform **boundary checks**, making them vulnerable to buffer overflows.

Unsafe Example:

```
char buffer[10];
strcpy(buffer, "This is a very long string!"); // Overflows buffer
```

Safer Alternative:

```
strncpy(buffer, "This is a very long string!", sizeof(buffer) - 1);
buffer[sizeof(buffer) - 1] = '\0'; // Ensuring null termination
```

Security Impact: Attackers can **overwrite function return addresses**, **manipulate execution flow**, or cause a **denial-of-service (DoS)** condition.

Prevention: Use safer alternatives like **strncpy**, **snprintf**, and **input validation** to prevent buffer overflows.

Race Conditions

A **race condition** occurs when two or more processes compete for a shared resource within the same time interval, leading to **inconsistent, undesired, or incorrect outcomes**. These conditions arise in **operating systems, multithreaded applications, or concurrent processes**, where the execution order of competing instructions affects the final result.

How Race Conditions Occur

- A typical scenario involves **two processes trying to access the same memory chunk**:
    1. Process A checks if a resource is available.
    2. Process B does the same before A claims the resource.
    3. Both processes receive a **"yes"**, but only one gets the resource, leading to an error.
- This flaw affects **data integrity and process correctness**, making it a critical issue in **system security**.

Security Implications of Race Conditions

- **Integrity failure**: A program may produce unpredictable or **incorrect results** due to concurrent execution issues.
- **Unauthorized file access or modification**: Attackers can manipulate race conditions to **alter system files or gain elevated privileges**.
- **Exploitation Example – Tripwire Vulnerability**:
    - The security tool **Tripwire** used temporary files to log activity.
    - The original process:
        1. Tripwire generates a filename for logging.
        2. It checks if the file exists.
        3. It creates the file.
        4. Later, it writes logs to the file.
    - **Attack Scenario**: A malicious process can replace the temporary file with a **symbolic link to another system file** before Tripwire writes data, effectively **overwriting sensitive files**.
    - This demonstrates how an **attacker can manipulate a race condition between file creation and access** to modify critical system files.

Mkdir Race Condition Attack

A **mkdir race condition attack** occurs when an attacker exploits the **timing gap between space allocation and permission assignment** during the creation of a directory in Unix-like systems.

How the Attack Works:

1. **Mkdir allocates space**:
    - A privileged process (e.g., running as root) executes `mkdir` to create a directory.
    - The system **allocates disk space** for this directory but has not yet finalized access permissions.
2. **Attacker creates a symbolic link**:
    - Before the system assigns correct ownership/permissions, the attacker **replaces the allocated directory with a symbolic link** to a critical system file (e.g., `/etc/passwd`).
    - This happens because the allocation and permission-setting steps occur separately, creating a window of opportunity.
3. **System grants access**:
    - The operating system **completes the `mkdir` operation**, assuming the directory was created normally.
    - However, because the attacker replaced it with a symbolic link, **the privileged process unknowingly assigns ownership or modifies access permissions of a sensitive file** (e.g., `/etc/passwd`).

Security Impact

- The attacker **gains control over critical files**, potentially modifying system credentials or injecting malicious configurations.
- They can **escalate privileges**, leading to **unauthorized access** or a complete system compromise.
- This technique is especially dangerous if used against **automated scripts or privileged processes** that create directories in a predictable manner.

Preventing Race Conditions

Since **race conditions are harder to exploit than buffer overflows**, they are not as commonly targeted. However, securing against them requires **making security-critical processes atomic**—meaning they must occur **all at once, without interruption**.

Mitigation Strategies:

- **Use atomic operations**: Ensure **check-and-use actions happen simultaneously**.
- **Lock resources properly**: Use **mutexes or file locks** to prevent concurrent modifications.
- **Randomize file names**: Prevent attackers from guessing temporary file names.
- **Use secure system calls**: Functions like `open(O_CREAT | O_EXCL)` ensure **exclusive file creation**.
- **Implement race condition detection**: Security tools can analyze processes for possible **timing-based vulnerabilities**.

Malware

Malware refers to **malicious software** designed to **disrupt, damage, or gain unauthorized access** to computer systems.

Types of Malware

Virus

- A program that **replicates itself** and infects other non-malicious programs by **modifying them**.
- Requires a **host program** to spread.

Worm

- A self-replicating program that **spreads copies of itself through a network**.
- Does **not require a host program** to spread.

Trojan Horse

- A program that appears to perform a **legitimate function** but contains **hidden malicious functionality**.
- Often used to **steal data, create backdoors, or execute harmful actions**.

Worms and Trojans have been mentioned in the syllabus, but no content has been provided in the notes.

Worm

A worm is a type of malware that spreads across networks by exploiting security vulnerabilities or using social engineering techniques. Unlike viruses, worms do not need a host file or user intervention to replicate; they spread autonomously by scanning for and infecting vulnerable devices. Once inside a system, a worm can consume bandwidth, slow down network performance, and facilitate further attacks, such as dropping additional malware or creating backdoors for remote control. Notable examples include the **Morris Worm (1988)** and the **WannaCry ransomware worm (2017)**, which caused widespread disruption by self-propagating across the internet.

Trojan Horse

A Trojan horse is a deceptive piece of software that masquerades as a legitimate application to

trick users into installing it. Once executed, it can perform various malicious actions, such as stealing sensitive information, creating unauthorized access points, or delivering additional malware. Unlike worms, Trojans do not self-replicate but rely on users to install them, often through phishing emails, fake downloads, or malicious attachments. For example, the **Zeus Trojan** was used to steal banking credentials, while the **Emotet Trojan** served as a malware delivery tool for other cyber threats.

Harm from Malicious Code

Harm to users and systems:

•Sending email to user contacts
•Deleting or encrypting files
•Modifying system information, such as the Windows registry
•Stealing sensitive information, such as passwords
•Attaching to critical system files
•Hide copies of malware in multiple complementary locations

Harm to the world:

•Some malware has been known to infect millions of systems, growing at a geometric rate
•Infected systems often become staging areas for new infections

Viruses:

A virus is a type of malicious software that attaches itself to a legitimate program or file and spreads by modifying other programs when the infected file is executed. Unlike worms, viruses require user action to propagate, such as opening an infected email attachment or running a compromised application. Once activated, a virus can corrupt files, disrupt system performance, steal data, or render a system inoperable. Some viruses are designed to be stealthy, while others cause immediate damage.

Transmission and Propagation of Viruses

- **Setup and Installer Programs**
    - The SETUP program may call dozens or hundreds of other programs, some on the distribution medium, some already residing on the computer, some in memory.
    - If any one of these programs contains a virus, the virus code could be activated and could install itself on a permanent storage medium (typically, a hard disk)
    - Could install itself in any and all executing programs in memory
- **Attached Files**
    - Virus writers trick victims into opening infected files.
    - Once opened/executed, the virus activates and spreads.
- **Document Viruses**
    - Embedded in formatted documents (e.g., Word files, databases, spreadsheets).
    - The virus executes when the document is opened.
- **Autorun Viruses**
    - Autorun is a feature of operating systems that causes the automatic execution of code based on name or placement.
    - Early malicious code writers was to augment or replace autoexec.bat to get the malicious code executed.
- **Using Nonmalicious Programs**
    - **Appended Viruses**
        - Insert themselves into an executable before the first instruction.
        - Ensure activation when the program runs.
    - **Viruses that Surround a Program**
        - Control execution before and after running the original program.
        - Allow the virus to operate while maintaining normal program functionality.

Malware Activation

One-Time Execution (Implanting)

- Malicious code executes a one-time process to transmit, receive, and install the infection.

Boot Sector Viruses

- Bury the code among other system routines.
- Place the code on the list of programs started at computer startup to ensure reactivation.

Memory-Resident Viruses

- Resident code remains in memory after execution.
- Examples include routines that interpret keyboard inputs.

Application Files

- A virus macro can add itself to the startup directives of an application.
- Embeds itself in data files to spread infection to recipients.

Code Libraries

- Executing code in a library can spread the viral infection.
- Passes the infection to other transmission media.

Malware Detection

Three Common Methods

1. **Signature Detection**
2. **Change Detection**
3. **Anomaly Detection**

Each method has its advantages and disadvantages.

Signature Detection

- A signature is a unique string of bits (or hash value) found in software.
- Example: A virus may have a signature like `0x23956a58bd910345`.
- Scanning files for this signature can help detect known malware.
- However, signatures can appear in non-malicious files, leading to false positives.

Advantages:

- Effective against traditional malware.
- Minimal impact on users and administrators.

Disadvantages:

- Signature databases can be large, slowing down scanning.
- Requires frequent updates to detect new threats.
- Cannot detect unknown or evolving malware.

Change Detection

- Malware must reside somewhere on a system.
- Changes in files can indicate infection.
- Uses file hashing to detect modifications.
- Checks for oligomorphism or polymorphism in malware.

Advantages:

- Almost no false negatives.
- Can detect previously unknown malware.

Disadvantages:

- Many legitimate files change frequently, causing false positives.
- High maintenance burden on users and administrators.
- May still require signature-based verification for confirmation.

Anomaly Detection

- Monitors system behavior for unusual or potentially malicious activities.
- Detects changes in files, system performance, network activity, and file access.
- Requires a baseline definition of "normal" system behavior.

Advantages:

- Can detect unknown malware.

Disadvantages:

- Not fully reliable in practice.
- Attackers can disguise malicious behavior as normal by acting slowly.
- Needs to be combined with other methods (e.g., signature detection).
- Common in Intrusion Detection Systems (IDS), but remains a challenging problem.

Countermeasures

Countermeasures for Users

- Use software acquired from reliable sources.
- Test software in an isolated environment.
- Only open attachments when you know them to be safe.
- Treat every website as potentially harmful.
- Create and maintain backups.

Countermeasures for Developers

Software Engineering Techniques

- **Information Hiding** – Describe what a module does, not how it does it.
- **Modularity** – Use coupling and cohesion principles.
- **Mutual Suspicion** – Ensure components do not trust each other blindly.
- **Confinement** – Limit the spread of potential damage within a system.
- **Simplicity** – Complexity is often the enemy of security.
- **Genetic Diversity** – Diversity reduces the number of targets susceptible to a single attack type.

Testing

- **Unit Testing** – Checks if software components fulfill their functionalities.
- **Integration Testing** – Ensures smooth data flow between modules.
- **Function Testing** – Verifies that software features work as intended.
- **Performance Testing** – Measures system efficiency and speed.

- **Acceptance Testing** – Ensures software meets the requirements of a specification or contract.
- **Installation Testing** – Confirms proper installation and execution.
- **Regression Testing** – Ensures new updates do not break existing functionality.
- **Penetration Testing** – Simulates attacks to find security weaknesses.

Countermeasures Specifically for Security

Design Principles for Security

- **Least Privilege** – Grant only the minimum necessary permissions.
- **Economy of Mechanism** – Keep security mechanisms simple and efficient.
- **Open Design** – Security should not rely on secrecy.
- **Complete Mediation** – Every access attempt should be checked for authorization.
- **Permission-Based** – Access should be explicitly granted rather than assumed.
- **Separation of Privilege** – Require multiple conditions for granting access.
- **Least Common Mechanism** – Minimize shared resources to reduce attack surfaces.
- **Ease of Use** – Security mechanisms should be user-friendly.

Additional Security Measures

- **Penetration Testing for Security** – Identifies vulnerabilities through simulated attacks.
- **Proofs of Program Correctness** – Uses formal methods to verify that programs behave as expected.
- **Validation** – Ensures that inputs and system operations conform to expected behavior.
- **Defensive Programming** – Developers must not only write correct code but also anticipate potential misuse and errors.
- **Trustworthy Computing Initiative** – Ensures that all developers undergo security training and follow secure development practices.

Countermeasures That Don’t Work

- **Penetrate-and-Patch** –

  "Penetrate and patch" is a security strategy that involves attacking a system to find vulnerabilities, and then fixing those vulnerabilities.

  Ineffective because it is hurried, overlooks the context of the issue, and focuses on individual failures rather than system-wide security.

- **Security by Obscurity** – The false belief that a system can remain secure as long as its internal mechanisms are hidden from outsiders.