Chapter 3: Programs and Programming
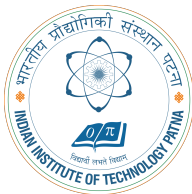
Dr. Mayank Agarwal

Department of CSE
IIT Patna

CS457 Big Data Security
Jan-April 2026

- Moving from Theory to Practice
- How Security Flaws Become Embedded in Software
- The Programmer's Role in Security

We now apply our security toolbox to the environment where vulnerabilities are most often created: software. This chapter examines why programs fail, how attackers exploit those failures, and what developers can do to write more secure code. The bugs we discuss here are the root cause of most real-world attacks.

Chapter 3 Learning Objectives

- Understand the nature and causes of program security vulnerabilities.
- Learn specific, common code flaws: buffer overflows, injection attacks, race conditions.
- Comprehend malicious code: viruses, worms, trojans, and ransomware.
- Study countermeasures: secure development practices, testing, and runtime protection.
- Recognize the programmer's critical role in system security.

This chapter connects the abstract security goals from Chapter 1 and the tools from Chapter 2 to the concrete reality of buggy code. You will learn to think like both a defender (how to prevent flaws) and an attacker (how to find and exploit them).

The Problem: Why Are Programs Insecure?

- **Complexity:** Modern programs are enormously complex (millions of lines of code).
- **Human Fallibility:** Programmers make mistakes. Security is often an afterthought.
- **Legacy Code:** Systems built without security in mind, now critical to operations.
- **Environmental Dependence:** Programs run in untrusted, hostile environments (the internet).
- **Economic Pressures:** Time-to-market often prioritized over security.

Writing correct code is hard. Writing securely correct code is even harder. It requires anticipating not just how users will use the program, but how adversaries will *abuse* it. This adversarial mindset must be integrated into the development process.

The Vulnerability Lifecycle in Software

- **Introduction:** A developer creates a flaw during design or coding.
- **Deployment:** The flawed code is distributed and executed.
- **Discovery:** The flaw is found (by researchers, attackers, or users).
- **Disclosure:** The flaw is reported (privately to vendor or publicly).
- **Exploit:** Attackers write code to take advantage of the flaw.
- **Patch:** The vendor develops and releases a fix.
- **Resolution:** Users apply the patch; the window of exposure closes.

The goal of secure programming is to prevent introduction. The goal of defensive operations is to shrink the window between Discovery and Resolution. Zero-day attacks exploit flaws between Discovery and Patch release.

Types of Program Security Vulnerabilities

- **Implementation Flaws:** Bugs in the code (buffer overflow, integer overflow).
- **Design Flaws:** Inherent problems in the architecture or logic (poor authentication scheme).
- **Environmental Flaws:** Misuse of or dependence on the runtime environment (unsafe library calls).
- **Configuration Flaws:** Insecure default settings or deployment errors.

Implementation flaws are the most common focus, but design flaws can be more devastating and harder to fix. A buggy login function can be patched; a fundamental design that transmits passwords in plaintext requires a complete rewrite.

Buffer Overflows: The "Classic" Vulnerability

- **Cause:** Writing data beyond the bounds of a fixed-size buffer in memory.
- **Mechanism:** Input is not properly checked for length. Excess data overwrites adjacent memory.
- **Consequences:** Can corrupt data, crash the program, or, critically, **allow execution of attacker-provided code**.
- Most prevalent in languages like C and C++ that allow direct memory access without automatic bounds checking.

Buffer overflows have been the "king" of vulnerabilities for decades. They directly violate the *Fail-Safe Defaults* principle—the program assumes input will fit, rather than checking. They enable attackers to hijack a program's control flow.

Anatomy of a Stack-Based Buffer Overflow

- The **stack** stores local variables, function parameters, and return addresses.
- A buffer declared as a local variable resides on the stack.
- If overflowed, adjacent memory is overwritten, including the **saved return address**.
- Attacker crafts input so the overflow overwrites the return address with a pointer to their own malicious code (shellcode) placed in the buffer.
- When the function returns, it jumps to and executes the shellcode.

This is a simplified model. Modern defenses (stack canaries, DEP, ASLR) make classic stack overflows harder, but not impossible. Understanding this is crucial to understanding those defenses.

Heap-Based Overflows and Other Memory Corruption

- **Heap:** Dynamically allocated memory (via `malloc`, `new`).
- Overflowing a heap buffer can corrupt heap management data structures.
- Can lead to arbitrary code execution or application crash.
- **Off-by-One Errors:** Writing one byte past the buffer boundary. Can be just as exploitable.
- **Format String Vulnerabilities:** Using unsanitized user input as the format string for `printf`, `sprintf`. Can lead to reading or writing arbitrary memory.

Heap exploitation is more complex than stack exploitation but remains a serious threat. Format string bugs are a different class of flaw where the attacker controls the *format specifier*, not just the data.

Integer Overflows and Underflows

- **Integer Overflow:** An arithmetic operation results in a value larger than the maximum size the integer type can hold. It "wraps around" to a small value.

- **Integer Underflow:** Result is smaller than the minimum value, wrapping to a large value.

- Often leads to buffer overflows: A length check passes based on the wrapped value, but the actual memory allocation or copy uses the true, large value.

These are subtle logic errors. They happen because programmers assume arithmetic results will be sensible. They violate *Complete Mediation*—the check (on the wrapped value) is not a true mediation of the actual operation.

Injection Attacks

- **Concept:** Attacker supplies input that is interpreted as *code* or *commands* by a vulnerable program.
- The program fails to distinguish between **data** and **control information**.
- **Common Types:**
  - SQL Injection
  - Command Injection (OS Commanding)
  - Script Injection (Cross-Site Scripting - XSS)
  - LDAP Injection, XPath Injection

Injection is arguably the most critical web vulnerability. It stems from the fundamental security error of trusting user input. The program constructs a string (a query, command, script) by concatenating user input without proper sanitization.

SQL Injection in Detail

- **Vulnerable Code:** `sql = "SELECT * FROM users WHERE name = '" + userName + "'";`
- **Malicious Input:** `userName = "admin' OR '1'='1"`
- **Resulting Query:** `SELECT * FROM users WHERE name = 'admin' OR '1'='1'`
- The `OR '1'='1'` makes the WHERE clause always true, returning all users.
- More advanced attacks can modify data (`UPDATE`, `DELETE`), drop tables, or even execute commands on the database server.

SQL injection exploits the trust between the application and the database. The application sends a string; the database interprets it. If the attacker can control part of that string, they control part of the query's logic.

Command Injection

- **Vulnerable Code (PHP):** `system("ping -c 4 " . $_GET['host']);`
- **Malicious Input:** host = 8.8.8.8; cat /etc/passwd
- **Result:** The system executes: `ping -c 4 8.8.8.8; cat /etc/passwd`
- The semicolon allows multiple commands. The attacker's command runs with the program's privileges.

Command injection is often catastrophic because it gives the attacker direct shell access. It violates *Least Privilege*—why does a network diagnostic function need the ability to read the password file? It also fails to validate or escape input.

Cross-Site Scripting (XSS)

- A web application vulnerability where attacker-controlled script is injected into pages viewed by other users.
- **Reflected XSS:** Malicious script is part of the victim's request (e.g., in a URL parameter) and is immediately reflected back in the response.
- **Stored XSS:** Malicious script is stored on the server (e.g., in a forum post) and served to multiple victims.
- **DOM-based XSS:** Vulnerability exists in client-side JavaScript that manipulates the DOM with untrusted data.

XSS breaks the trust between a user and a website. The user's browser trusts scripts from the legitimate site. If an attacker can inject their script, it runs in that trusted context, allowing theft of cookies, session hijacking, or defacement.

Race Conditions (TOCTTOU)

- **Time-of-Check to Time-of-Use (TOCTTOU):** A program checks a condition (e.g., file permissions) and later uses the result, but the condition can change between the check and the use.

- **Example:** A setuid program checks if a user has read access to a file, opens it later. An attacker swaps the file with a symbolic link to a sensitive file between the check and the open.

- Exploits the gap between a check and an action. A violation of *Complete Mediation*—the check is not atomic with the use.

Race conditions are concurrency flaws. They are subtle, non-deterministic, and hard to reproduce. They often occur in privileged programs and can lead to privilege escalation. The fix is to make the check and use an atomic operation.

Incomplete Mediation

- When a program fails to validate all aspects of its input.
- **Example:** A web application accepts a numeric "price" parameter from a hidden form field: <input type="hidden" name="price" value="100">. An attacker changes it to price=-10 before submitting.
- The server-side code trusts the client-provided value without re-validating business logic (e.g., that price must be positive).
- **Never trust the client!** Client-side validation is for usability, not security.

This is a direct violation of the *Complete Mediation* principle. All inputs, from all sources (users, networks, files, other programs) must be validated for type, length, format, and business rules before use.

Malicious Code: Viruses

- **Definition:** A program that can infect other programs by modifying them to include a (possibly evolved) copy of itself.
- **Key Characteristic:** Requires human action to spread (e.g., running an infected program, opening a document).
- **Components:** Infection mechanism, trigger (payload), payload (malicious action).
- **Types:** File infector, boot sector, macro, script.

The classic "virus" metaphor is apt: it attaches itself to a host. Its spread is parasitic and requires a host program. The payload may be destructive (deleting files) or subtle (stealing data). Defense focuses on detection (antivirus) and prevention (user education).

Malicious Code: Worms

- **Definition:** A standalone program that replicates itself to spread to other computers, **without human intervention**.
- Exploits network-based vulnerabilities (e.g., buffer overflows, weak passwords) to propagate automatically.
- **Components:** Target discovery, infection mechanism, payload.
- Causes damage through bandwidth consumption, system instability, and payload execution.
- **Examples:** Morris Worm (1988), Code Red, SQL Slammer, Conficker.

Worms are self-replicating malware. They are a major threat because of their speed and autonomy. The SQL Slammer worm infected 75,000 systems in 10 minutes in 2003. Defense requires patching vulnerabilities, network segmentation, and intrusion detection.

Malicious Code: Trojan Horses

- **Definition:** A malicious program disguised as, or embedded within, legitimate, useful software.
- Does not replicate itself. Spreads by user deception (social engineering).
- **Purpose:** Creates a backdoor, steals information, downloads other malware.
- **Example:** A "free" game or "system optimizer" that secretly installs a keylogger.

The name comes from the Greek story. The user willingly installs the software, not knowing its true intent. This bypasses many technical controls. Defense relies heavily on user education, code signing, and application whitelisting.

Malicious Code: Ransomware

- **Definition:** Malware that encrypts the victim's files and demands a ransom (in cryptocurrency) for the decryption key.
- Combines the propagation methods of worms/trojans with a destructive, profit-driven payload.
- Causes massive disruption to businesses, hospitals, and governments.
- **Examples:** WannaCry (used a worm component), CryptoLocker, Ryuk.

Ransomware is a business model for criminals. It directly attacks availability. Defense requires a multi-layered approach: patching, backup and recovery plans (offline backups!), user training, and endpoint protection.

Other Malware: Spyware, Rootkits, Logic Bombs

- **Spyware:** Covertly gathers information about a user/org (keystrokes, browsing habits).
- **Rootkit:** A set of tools that provides persistent, stealthy access to a system, often by modifying the OS kernel or system utilities to hide its presence.
- **Logic Bomb:** Malicious code that lies dormant until a specific condition is met (date, event, command), then triggers a payload.

These categories often overlap. A trojan may install a rootkit to hide itself, which includes spyware. A logic bomb could be planted by a disgruntled employee. Rootkits are particularly dangerous because they subvert the OS's ability to report truthfully.

Countermeasure Principle: Secure Development Lifecycle (SDL)

- Integrate security throughout the entire software development process.
- **Phases:**
  1. Requirements (define security requirements)
  2. Design (threat modeling, architecture review)
  3. Implementation (secure coding standards, code reviews)
  4. Verification (security testing: SAST, DAST, fuzzing)
  5. Release (incident response plan)
  6. Response (patch development)

Fixing bugs after release is 100x more expensive than preventing them during design. The SDL is a process to institutionalize security thinking. Microsoft's SDL is a famous, successful example that dramatically reduced vulnerabilities in their products.

Countermeasure: Defensive Programming

- A mindset and set of coding practices to assume that failures will happen and inputs are hostile.
- **Key Practices:**
    - Validate all input (check type, length, range, format).
    - Use safe library functions (e.g., `strncpy` instead of `strcpy`).
    - Handle errors gracefully; fail securely (default deny).
    - Principle of least privilege for program components.
    - Keep it simple (Economy of Mechanism).

This is the programmer's application of the security mindset. Never say, "This should never happen." Code for the case where it *does* happen. Input validation is the single most important defensive programming practice.

Input Validation and Sanitization

- **Validation (Whitelisting):** Define what is *good* and reject everything else. Most secure.
- **Sanitization (Cleaning):** Transform input to make it safe (e.g., escape metacharacters).
- For structured data (SQL, OS commands), use **parameterized interfaces**.
- **SQL Example:** Use Prepared Statements: `stmt = conn.prepareStatement("SELECT * FROM users WHERE name = ?"); stmt.setString(1, userName);`
- This separates code from data. The database knows `userName` is always a data value, never part of the SQL syntax.

Parameterized queries/prepared statements are the *only* reliable defense against SQL injection. Escaping is fragile and error-prone. Similarly, for OS commands, avoid them entirely, or use APIs that pass arguments as a list, not a concatenated string.

Countermeasure: Compile-Time and Runtime Protections

- **Compiler Flags:** `-fstack-protector` (stack canaries), `-D_FORTIFY_SOURCE=2` (fortify source).
- **Data Execution Prevention (DEP) / NX Bit:** Marks memory pages as non-executable, preventing code execution from stack/heap.
- **Address Space Layout Randomization (ASLR):** Randomizes memory addresses of key areas (stack, heap, libraries) making it hard for attackers to predict target addresses.
- **Control Flow Integrity (CFI):** Ensures program execution follows a valid path determined at compile time.

These are OS/hardware/compiler defenses that make exploitation harder. They are not silver bullets but raise the bar. DEP+ASLR is the standard baseline. They exemplify defense in depth—even if a buffer overflow occurs, exploitation may be blocked.

Safe Languages and Memory Safety

- **Memory-safe languages:** Java, C#, Python, Rust, Go.
- Automatically manage memory (garbage collection) and perform bounds checking on array/string accesses.
- Eliminate entire classes of vulnerabilities (buffer overflows, use-after-free, double-free).
- **Trade-off:** Often have performance overhead and less low-level control than C/C++.

Using a memory-safe language is one of the most effective ways to prevent memory corruption vulnerabilities. Rust is notable because it provides memory safety *without* garbage collection, using a compile-time ownership model. This is a paradigm shift in systems programming.

Code Analysis and Testing

- **Static Application Security Testing (SAST):** Analyzes source code for vulnerabilities without running it. Good for finding coding standard violations, potential bugs early.

- **Dynamic Application Security Testing (DAST):** Tests the running application (e.g., a web app) by sending malicious inputs. Finds runtime vulnerabilities like injection.

- **Fuzzing (Fuzz Testing):** Provides random, malformed, or unexpected inputs to a program to trigger crashes or uncover vulnerabilities. Highly effective for finding memory corruption bugs.

SAST is like a spell-checker for security bugs. DAST is like a penetration test. Fuzzing is like stress-testing with chaos. They are complementary. All should be part of a mature development process.

Operating System Controls for Program Security

- **Principle of Least Privilege:** Run programs with the minimum privileges needed (e.g., not as root/Administrator).

- **Sandboxing:** Isolate a program from the rest of the system, limiting its access to resources (files, network). Examples: Containers (Docker), JavaScript sandbox in browser, app sandboxing on mobile OS.

- **Mandatory Access Control (MAC) Systems:** SELinux, AppArmor enforce system-wide policy on what processes can do, beyond user permissions.

If you can't make the program perfectly secure, contain its potential damage. Sandboxing implements *Least Common Mechanism*—if the program is compromised, the attacker is trapped in the sandbox. This is crucial for handling untrusted code (e.g., browser plugins, downloaded apps).

The Human Element: Training and Culture

- Developers must be trained in secure coding practices.
- Security teams and development teams must collaborate, not be adversaries.
- Management must prioritize and reward secure code, not just features and speed.
- **Shift Left:** Integrate security earlier in the development lifecycle.

Technology alone fails. A developer who doesn't understand SQL injection will write vulnerable code, no matter what tools are available. Building a "security-aware" engineering culture is the most important, and most difficult, long-term countermeasure.

Case Study: The Morris Worm (1988)

- One of the first major internet worms.
- Exploited several vulnerabilities: buffer overflow in `fingerd`, weak passwords, debug mode in sendmail.
- Caused massive disruption. Led to the creation of the first CERT (Computer Emergency Response Team).
- **Lessons:** The danger of homogeneous software, the importance of patching, the need for coordinated response.

The Morris Worm was a wake-up call. It demonstrated how interconnected systems could be used for rapid, automated attack propagation. Many of the vulnerabilities it exploited are still common today (weak passwords, unpatched services).

Case Study: Heartbleed (OpenSSL Vulnerability, 2014)

- A buffer over-read bug in the OpenSSL implementation of the TLS heartbeat extension.
- Allowed attackers to read up to 64KB of *server memory* per request—potentially exposing private keys, session cookies, user data.
- **Cause:** Missing bounds check on a `memcpy` operation. A classic implementation flaw in a critical security library.
- **Impact:** Hundreds of thousands of servers vulnerable. Required mass reissuance of certificates and keys.

Heartbleed shows that even security-critical code written by experts in a "safe" language (C) is vulnerable to memory bugs. It also highlights the risk of widespread software dependencies (OpenSSL is in everything).

Emerging Threats: Supply Chain Attacks

- Compromising software *before* it reaches the user, by attacking the development or distribution process.
- **Methods:** Hacking build servers, compromising open-source libraries, poisoning software updates.
- **Examples:** SolarWinds Orion compromise (2020), malicious npm/PyPi packages.
- **Defense:** Code signing, reproducible builds, vetting of third-party dependencies, integrity checks.

This shifts the attack upstream. You can have perfect secure coding practices, but if an attacker replaces your compiler or a library you depend on, your software is compromised. This violates trust in the entire software ecosystem.

Summary: The Program Security Landscape

- Vulnerabilities stem from **implementation flaws** (buffer overflows, injection), **design flaws**, and **malicious code**.
- Exploitation turns these flaws into loss of confidentiality, integrity, or availability.
- Countermeasures must be applied throughout the **development lifecycle** (SDL), using **defensive programming**, **safe languages**, **runtime protections**, and **operating system controls**.
- Ultimately, security depends on **skilled, aware developers** and a supportive **organizational culture**.

Program security is a vast field. This chapter provides the taxonomy and foundational knowledge. The key is to understand that vulnerabilities are predictable and preventable. The attacker's methods are your checklist for defense.

Looking Ahead

- **Chapter 4: The Web – User Side** – We will apply these program security concepts to the client-side environment of the web browser.
- **Chapter 5: The Web – Server Side** – We will examine server-side web application security in depth.
- The web is the primary attack surface today, built entirely on programs.

The principles you've learned here—input validation, memory safety, least privilege—are directly applicable to the web. In the next chapters, we'll see how failures in web programs lead to data breaches, account takeovers, and compromised sites.

Critical Thinking Questions (1)

- Why does using a memory-safe language like Java or Python not automatically eliminate all security vulnerabilities in a program? Give an example of a vulnerability that could still exist.

- Compare and contrast SQL Injection and Cross-Site Scripting (XSS). What is the fundamental cause they share? How do their impacts differ?

Memory safety stops memory corruption, but logic flaws, injection, and design flaws remain. SQLi and XSS both stem from unsanitized input, but SQLi attacks the database/server, while XSS attacks other users of the web application.

Critical Thinking Questions (2)

- A developer fixes a SQL injection vulnerability by writing a function that escapes single quotes (') in user input before putting it into a query. Explain why this is a fragile defense compared to using parameterized queries.

- Explain how Address Space Layout Randomization (ASLR) makes exploiting a buffer overflow more difficult. What is a limitation of ASLR?

Escaping is context-dependent (different for SQL, OS, HTML) and easy to get wrong. Parameterization is robust. ASLR prevents the attacker from knowing where to jump (return address) but can be defeated by information leaks or if not all modules are randomized.

Hands-On Exercise Suggestions

- **Buffer Overflow:** Use a simple, vulnerable C program (with DEP/ASLR disabled for learning) to practice causing a crash and then redirecting control.

- **SQL Injection:** Set up a vulnerable web app (e.g., OWASP WebGoat, Damn Vulnerable Web App) and practice extracting data, bypassing login, and using UNION attacks.

- **Code Review:** Take a small piece of code (in C or a web language) and perform a manual security review, looking for the vulnerabilities discussed in this chapter.

There is no substitute for hands-on experience with vulnerabilities in a controlled lab environment. It transforms abstract concepts into concrete understanding. Always do this in a isolated VM or dedicated lab, never on production or unauthorized systems.

Chapter 3: Key Takeaways

- Most security breaches start with a software vulnerability.
- Buffer overflows and injection attacks are among the most critical and common flaws.
- Malicious code (viruses, worms, trojans) exploits these flaws or uses deception.
- Prevention requires a lifecycle approach: secure design, defensive programming, safe languages, and runtime protections.
- The programmer is the first and most important line of defense.

You now understand the "how" behind many headlines about data breaches and cyber attacks. More importantly, you understand the mindset and practices needed to prevent them. This knowledge is essential whether you become a developer, an auditor, or a security analyst.

Secure Coding is Not an Option; It's a Requirement.

Next: Chapter 4 - The Web – User Side

The web is the battlefield where these program vulnerabilities are most actively exploited. In the next chapter, we examine the security environment from the user's perspective—the browser, cookies, and client-side attacks.