Never Trust the User
Secure Coding and Input Validation

Brendan Shea, PhD

Programming and Problem Solving • Rochester Community and Technical College

Introduction

In 1988, a graduate student named Robert Morris released a program that brought roughly 10% of the entire Internet to its knees. The **Morris Worm** exploited several vulnerabilities, including a classic input validation flaw: a program that copied user input into memory without checking how much data it received. Morris's "experiment" caused millions of dollars in damage and earned him the first conviction under the Computer Fraud and Abuse Act.

The lesson from 1988 remains the cardinal rule of secure programming: **Never trust the user.** Every piece of data entering your program—from keyboard input to file uploads to network messages—is potentially malicious. Attackers have spent decades crafting inputs designed to make programs behave in unintended ways, and they've succeeded spectacularly: data breaches, stolen passwords, crashed systems, and compromised infrastructure.

This case study explores the art of **input validation**—checking that data meets your expectations before using it. You'll learn why validation matters, how attackers exploit its absence, and how to write code that defends itself.

> **The Fundamental Principle**
>
> **All input is guilty until proven innocent.** Whether it comes from a user typing at a keyboard, a file on disk, a network connection, or another program, you must validate data before trusting it. The consequences of failing to do so range from crashed programs to catastrophic security breaches.

1 A Brief History of Things Going Wrong

Understanding security requires understanding how attackers think. Let's examine some landmark incidents that shaped our understanding of input validation.

> **The Morris Worm (1988)**
>
> Robert Morris, a Cornell graduate student and son of a chief scientist at the NSA, released a self-replicating program intended to gauge the size of the Internet. It exploited buffer

overflows in the `fingerd` and `sendmail` programs—services that blindly copied user input without checking its length. The worm spread faster than anticipated, reinfecting machines and consuming resources until systems ground to a halt. Morris was fined $10,050, sentenced to probation, and required to perform community service. He later became a professor at MIT. The incident led to the creation of CERT (Computer Emergency Response Team), the first organization dedicated to coordinating responses to security incidents.

SQL Injection Attacks (2000s–Present)

In 2008, a SQL injection attack compromised Heartland Payment Systems, exposing 130 million credit card numbers—the largest data breach in history at that time. The attacker inserted malicious database commands through a web form that failed to validate its input. SQL injection remains in the OWASP Top 10 security vulnerabilities decades after it was first documented, because developers keep making the same mistake: constructing database queries by concatenating user input.

The Equifax Breach (2017)

Attackers exploited a vulnerability in Apache Struts, a web application framework, to breach Equifax. The flaw allowed attackers to execute arbitrary code through maliciously crafted input. The breach exposed personal information of 147 million people—Social Security numbers, birth dates, addresses, and driver's license numbers. Equifax had failed to patch a known vulnerability for months, but the underlying issue was the same: insufficient input validation in the framework's code.

Log4Shell (2021)

A vulnerability in Log4j, a ubiquitous Java logging library, allowed attackers to execute code simply by sending specially crafted text strings. If a vulnerable application logged user input (like a username or search query), the attacker's payload would execute. The vulnerability, called Log4Shell, affected millions of applications worldwide—from Minecraft servers to Apple's iCloud. It demonstrated that even "safe" operations like logging can be dangerous if input isn't properly sanitized.

These incidents share a common thread: programs that accepted data without adequate scrutiny. The specific mechanisms differ, but the fundamental failure is the same.

2 Why Java Is Safer (But Not Safe)

Java was designed in the 1990s with security in mind, learning from decades of vulnerabilities in C and C++. Several language features provide protection that C programmers must implement manually—or more often, forget to implement.

2.1 Memory Safety and Bounds Checking

In C, arrays are just pointers to memory. Nothing stops you from reading or writing past the end of an array:

C: No Bounds Checking (Dangerous!)

```
char buffer[10];
// C allows this -- writes past the end of the array!
strcpy(buffer, "This string is way too long for the buffer");
// Memory corruption: other variables, return addresses overwritten
```

This is a **buffer overflow**—the foundation of countless exploits including the Morris Worm. Attackers craft input that overflows a buffer and overwrites critical data structures, potentially gaining control of the program.

Java prevents this entirely:

Java: Automatic Bounds Checking

```
int[] array = new int[10];
array[15] = 42;  // Throws ArrayIndexOutOfBoundsException
// Program crashes safely rather than corrupting memory
```

Every array access in Java is checked at runtime. If you try to access an invalid index, Java throws an exception rather than corrupting memory. This single feature eliminates an entire class of vulnerabilities.

C: Buffer Overflow

Java: Exception Thrown

buffer[3]

your array

corrupted!

other data

✗

blocked!
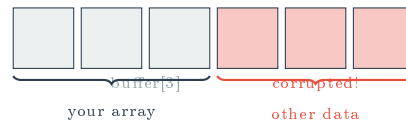
2.2   No Pointer Arithmetic

In C, you can manipulate memory addresses directly, pointing to arbitrary locations. Java has references but no pointer arithmetic—you cannot calculate memory addresses or access memory that wasn't properly allocated.

2.3   Automatic Memory Management

C requires programmers to manually allocate and free memory. Mistakes lead to "use-after-free" vulnerabilities—accessing memory that's been returned to the system and might now contain attacker-controlled data. Java's garbage collector handles memory automatically, eliminating this class of bugs.

2.4   The Catch: Logic Bugs Remain

Java's safety features prevent low-level memory corruption, but they cannot prevent logic errors. If your code trusts user input when it shouldn't, Java won't save you:

Java: Still Vulnerable to Logic Bugs

```
// User provides their "age"
String input = scanner.nextLine();
int age = Integer.parseInt(input);
```

```
// What if input is "-5"? "999"? "0"?
// Java won't stop you from using an invalid age
if (age >= 18) {
    grantAccess();  // Attacker enters "999", bypasses check
}
```

This is where input validation comes in.

3   Types of Input-Based Attacks

Attackers have developed numerous techniques for exploiting insufficient input validation. Understanding these attacks helps you defend against them.

3.1   SQL Injection

SQL injection occurs when user input is incorporated directly into database queries. Imagine a login form:

Vulnerable: String Concatenation in SQL

```
String username = scanner.nextLine();
String password = scanner.nextLine();

// DANGEROUS: Building query with user input
String query = "SELECT * FROM users WHERE name='" + username
               + "' AND password='" + password + "'";
```

If a user enters the username **admin' -**, the query becomes:

The Resulting Query

```
SELECT * FROM users WHERE name='admin' --' AND password=''
```

The - is a SQL comment, so everything after it is ignored. The attacker logs in as admin without knowing the password! Even worse, an attacker could enter **'; DROP TABLE users; -** and delete your entire user database.

Defense: Parameterized Queries

Never build SQL queries by concatenating strings. Use parameterized queries (prepared statements) that separate code from data:

```
        PreparedStatement ps = conn.prepareStatement(
    "SELECT * FROM users WHERE name=?  AND password=?");
                ps.setString(1, username);
                ps.setString(2, password);
```

The database treats parameters as pure data, never as SQL commands.

3.2   Command Injection

Similar to SQL injection, **command injection** occurs when user input is passed to system commands:

Vulnerable: Command Injection

```
String filename = scanner.nextLine();
// DANGEROUS: User input in system command
Runtime.getRuntime().exec("ls " + filename);

// Attacker enters: "file.txt; rm -rf /"
// Command becomes: ls file.txt; rm -rf /
// Second command deletes everything!
```

3.3   Cross-Site Scripting (XSS)

In web applications, **cross-site scripting** occurs when user input is displayed on a page without escaping. An attacker might enter their "name" as:

```
<script>stealCookies()</script>
```

If the website displays this without sanitization, the script executes in other users' browsers, potentially stealing their session cookies or credentials.

3.4   Integer Overflow

Even numeric input can be dangerous. What happens when arithmetic exceeds the maximum value a type can hold?

Integer Overflow

```
int quantity = scanner.nextInt();   // User enters 2,000,000,000
int price = 10;
int total = quantity * price;       // Overflow! Result is negative

if (total > 0) {
    chargeCustomer(total);  // Doesn't execute -- total is negative!
}
// Attacker gets items for free (or gets paid!)
```

4   Input Validation in Practice

Now let's see how to validate input properly. The goal is to reject bad data before it can cause harm.

4.1   Validating Numeric Input

The `Scanner` class can throw exceptions if users enter non-numeric data when you expect numbers:

Handling Non-Numeric Input

```java
Scanner scanner = new Scanner(System.in);
System.out.print("Enter your age: ");

int age;
if (scanner.hasNextInt()) {
    age = scanner.nextInt();
} else {
    System.out.println("Error: Please enter a number.");
    scanner.next();  // Clear the invalid input
    return;
}
```

But that's only half the battle—you must also check that the number is *reasonable*:

Complete Numeric Validation

```java
public static int getValidAge(Scanner scanner) {
    while (true) {
        System.out.print("Enter your age: ");

        // Check 1: Is it an integer?
        if (!scanner.hasNextInt()) {
            System.out.println("Please enter a whole number.");
            scanner.next();  // discard invalid input
            continue;
        }

        int age = scanner.nextInt();

        // Check 2: Is it in a valid range?
        if (age < 0) {
            System.out.println("Age cannot be negative.");
            continue;
        }
        if (age > 150) {
            System.out.println("Please enter a realistic age.");
            continue;
        }

        return age;  // Passed all checks!
    }
}
```

4.2   Validating String Input

String validation depends on what the string represents. For a username, you might check:
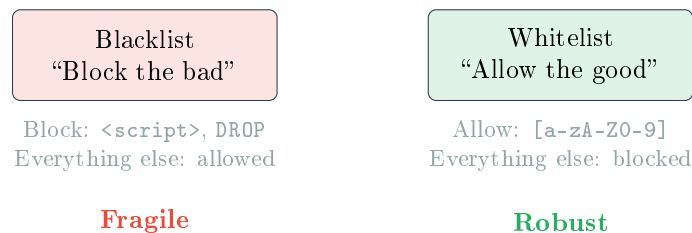
String Validation Example

```java
public static boolean isValidUsername(String username) {
    // Check 1: Not null or empty
    if (username == null || username.isEmpty()) {
        return false;
    }

    // Check 2: Length constraints
    if (username.length() < 3 || username.length() > 20) {
        return false;
    }

    // Check 3: Only allowed characters (letters, numbers, underscore
    )
    if (!username.matches("^[a-zA-Z0-9_]+$")) {
        return false;
    }

    // Check 4: Doesn't start with a number
    if (Character.isDigit(username.charAt(0))) {
        return false;
    }

    return true;
}
```

4.3   The Whitelist Approach
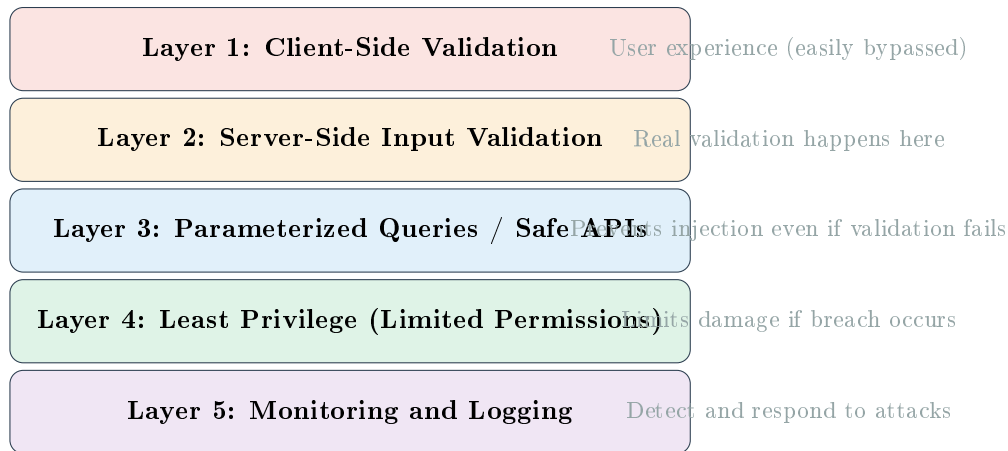
There are two philosophies for validation:

- **Blacklist**: Block known bad inputs (dangerous characters, malicious patterns)

- **Whitelist**: Allow only known good inputs (specific characters, formats, ranges)

Whitelisting is almost always safer. Blacklists are fragile—attackers constantly find new ways to encode malicious input that bypasses filters. With whitelisting, anything you haven't explicitly approved is rejected.

Blacklist "Block the bad"		Whitelist "Allow the good"
Block: `<script>`, `DROP` Everything else: allowed		Allow: `[a-zA-Z0-9]` Everything else: blocked
Fragile		**Robust**

5   Defense in Depth

No single validation check is foolproof. Security experts advocate **defense in depth**—multiple layers of protection so that if one fails, others still provide security.

Layer 1: Client-Side Validation   User experience (easily bypassed)

Layer 2: Server-Side Input Validation   Real validation happens here

Layer 3: Parameterized Queries / Safe APIs   Prevents injection even if validation fails

Layer 4: Least Privilege (Limited Permissions)   Limits damage if breach occurs

Layer 5: Monitoring and Logging   Detect and respond to attacks

Client-side validation (in the browser, before data is sent) improves user experience but provides *no security*—attackers can bypass it trivially. Real validation must happen on the server, where you control the code.

5.1 Fail Securely

When validation fails, your program must respond appropriately:

Failing Securely

```java
public static void processOrder(String productId, int quantity) {
    // Validate product ID
    if (!isValidProductId(productId)) {
        // Log the attempt (for security monitoring)
        logger.warn("Invalid product ID attempted: " + sanitize(
    productId));
        // Return generic error (don't reveal what went wrong)
        throw new InvalidInputException("Invalid request");
    }

    // Validate quantity
    if (quantity < 1 || quantity > MAX_ORDER_QUANTITY) {
        logger.warn("Invalid quantity attempted: " + quantity);
        throw new InvalidInputException("Invalid request");
    }

    // Only proceed if ALL validation passes
    executeOrder(productId, quantity);
}
```

Notice several security practices: logging suspicious input for later analysis, returning generic error messages (detailed errors help attackers understand your validation), and processing only after *all* checks pass.

6 Principles of Secure Input Handling

Let's summarize the key principles:

1. **Validate all input.** Every piece of data from outside your program—user input, files, network data, environment variables, command-line arguments—must be validated.

2. **Validate on the server.** Client-side checks are for user convenience only. An attacker controls the client and can send anything.

3. **Use whitelists.** Define what's allowed rather than trying to enumerate everything that's forbidden.

4. **Validate type, length, format, and range.** Is it the right data type? Is it within acceptable length limits? Does it match the expected format? Is the value in a sensible range?

5. **Reject, don't sanitize.** It's tempting to "fix" bad input by removing dangerous characters. This is risky—attackers find creative encodings that bypass sanitization. When possible, reject invalid input entirely.

6. **Use safe APIs.** Parameterized SQL queries, safe HTML encoding functions, and well-tested libraries handle dangerous operations correctly. Don't reinvent the wheel.

7. **Apply least privilege.** Even if an attacker bypasses validation, limit the damage by running with minimal permissions. A web application shouldn't have database admin rights.

8. **Log and monitor.** Track validation failures. Patterns of invalid input often indicate attack attempts. You can't respond to attacks you don't know about.

The cost of implementing these practices is small compared to the cost of a breach. Security isn't a feature you add at the end—it's a mindset you apply from the first line of code.

Discussion Questions

> ## Discussion Questions
>
> 1. **The Morris Worm Legacy:** Robert Morris exploited vulnerabilities in 1988, was convicted, and later became an MIT professor and successful entrepreneur. Some view him as a cautionary tale; others as a pioneer who exposed critical weaknesses. How should society treat security researchers who expose vulnerabilities through exploits? Is there a difference between malicious hacking and "research"?
>
> 2. **Language Design Trade-offs:** Java's bounds checking prevents buffer overflows but adds runtime overhead—every array access requires a check. C is faster but puts the burden on programmers. Is this trade-off worth it? Are there applications where C's approach is justified despite the risks?
>
> 3. **The Equifax Question:** Equifax had months to patch a known vulnerability but failed to do so. Who bears responsibility: the programmers who wrote the vulnerable code, the managers who didn't prioritize patching, the executives, or the company as a whole? How should liability for security breaches be assigned?
>
> 4. **Usability vs. Security:** Strict input validation can frustrate users. A name field that rejects "O'Brien" or "José" because of apostrophes and accents is both annoying and exclusionary. How do you balance security with usability and inclusion? What's the right approach for names, addresses, and other personal information?
>
> 5. **The Arms Race:** Attackers constantly develop new techniques; defenders patch and add validation; attackers find bypasses. Is this cycle inevitable? Could programming languages, frameworks, or development practices fundamentally change this dynamic, or will security always be a cat-and-mouse game?

Key Terms

Glossary of Key Terms

Buffer Overflow   A vulnerability where input exceeds allocated memory, overwriting adjacent data and potentially allowing code execution.

Bounds Checking   Verifying that array accesses are within valid indices; Java does this automatically, C does not.

Command Injection   An attack where malicious input is executed as system commands due to improper validation.

Cross-Site Scripting   A web vulnerability where attacker-supplied scripts execute in victims' browsers.

Defense in Depth   A security strategy using multiple protective layers so that failure of one doesn't compromise the system.

Input Validation   The process of verifying that data meets expected criteria before processing it.

Integer Overflow   When arithmetic exceeds a type's maximum value and wraps around, potentially causing logic errors.

Morris Worm   The 1988 self-replicating program that exploited buffer overflows and brought down much of the Internet.

Parameterized Query   A database query where user input is passed as parameters rather than concatenated into the query string.

SQL Injection   An attack where malicious SQL commands are inserted through improperly validated input fields.

Sanitization   Modifying input to remove or escape dangerous characters—generally less safe than rejection.

Whitelisting   Validating by defining allowed inputs; anything not explicitly permitted is rejected.