

Vulnerabilities II: Input Validation Errors and Defenses

Fall 2024

R. Sekar

What comes after buffer overflows?

- Most vulnerabilities reported in the early part of 2000s were due to memory corruption
 - Typically, 2/3rd to 4/5th of security advisories
- But things have changed dramatically since then
 - Web-related vulnerabilities dominate today
 - Increased use of web
 - Hybrid nature of web applications, with server and client-side components; and a mix of trusted/untrusted data
 - Less sophisticated developers

SQL Injection

- Attacker-provided data used in SQL queries:

```
$cmd = "SELECT price FROM products WHERE  
      name=' " . $name . " '"
```

... Use cmd as an SQL query

SQL Injection

- Attacker-provided data used in SQL queries:

```
$cmd = "SELECT price FROM products WHERE  
      name=' " . $name . " '"
```

... Use cmd as an SQL query

- Attacker-provided name:

```
xyz'; UPDATE products SET price=0 WHERE  
name='iphone15'
```

SQL Injection

- Attacker-provided data used in SQL queries:

```
$cmd = "SELECT price FROM products WHERE  
      name=' " . $name . " '"
```

... Use cmd as an SQL query

- Attacker-provided name:

```
xyz'; UPDATE products SET price=0 WHERE  
name='iphone15'
```

- Resulting query

```
SELECT price FROM products WHERE name='xyz';  
UPDATE products SET price=0 WHERE  
      name='iphone15'
```

Command Injection

- Attacker-provided data used in creation of command that is passed to the OS
- Example: SquirrelMail

```
$send_to_list = $_GET['sendto']
```

```
$command = "gpg -r $send_to_list 2>&1"
```

```
popen($command)
```

Command Injection

- Attacker-provided data used in creation of command that is passed to the OS

- Example: SquirrelMail

```
$send_to_list = $_GET['sendto']  
$command = "gpg -r $send_to_list 2>&1"  
popen($command)
```

- Attack: user fills in the following information in the “send” field of email:

```
xyz@abc.com; rm -rf *
```

Script Injection

- Similar to command injection: attacker-provided input used to create a string that is interpreted as a script
- Common in dynamic languages since these often allow string values to be eval'd
 - Most common web-application languages support eval: PHP, Python, Ruby, ...
- Format string attacks
 - Have similarity with script injection
 - The command language is that of format directives

Cross-Site Scripting

- **Attacker-provided data** controls scripts (i.e., Javascript code) on generated Web pages
- **Example:**
`http://www.xyzbank.com/findATM?zip=90100`

Cross-Site Scripting

- **Attacker-provided data** controls scripts (i.e., Javascript code) on generated Web pages

- **Example:**

`http://www.xyzbank.com/findATM?zip=90100`

- **Normal**

`<HTML>ZIP code not found: 90100 </HTML>`

Cross-Site Scripting

- **Attacker-provided data** controls scripts (i.e., Javascript code) on generated Web pages

- **Example:**

`http://www.xyzbank.com/findATM?zip=90100`

- **Normal**

`<HTML>ZIP code not found: 90100 </HTML>`

- **Attack**

`<HTML>ZIP code not found: <script
src="http://attacker.com/script.js"></script> </HTML>`

Directory traversal

- Attacker-provided path names contain directory traversal strings (e.g., “../”)

Directory traversal

- Attacker-provided path names contain directory traversal strings (e.g., "../")
- May be disguised by various encodings
- Example:

```
void check_access(char *file) {  
    if ((strstr(file, "/cgi-bin/") == file) &&  
        (strstr(file, "../") == NULL)) {  
        char *f = url_decode(file);  
        /* allow access to f ... */  
    }  
}
```

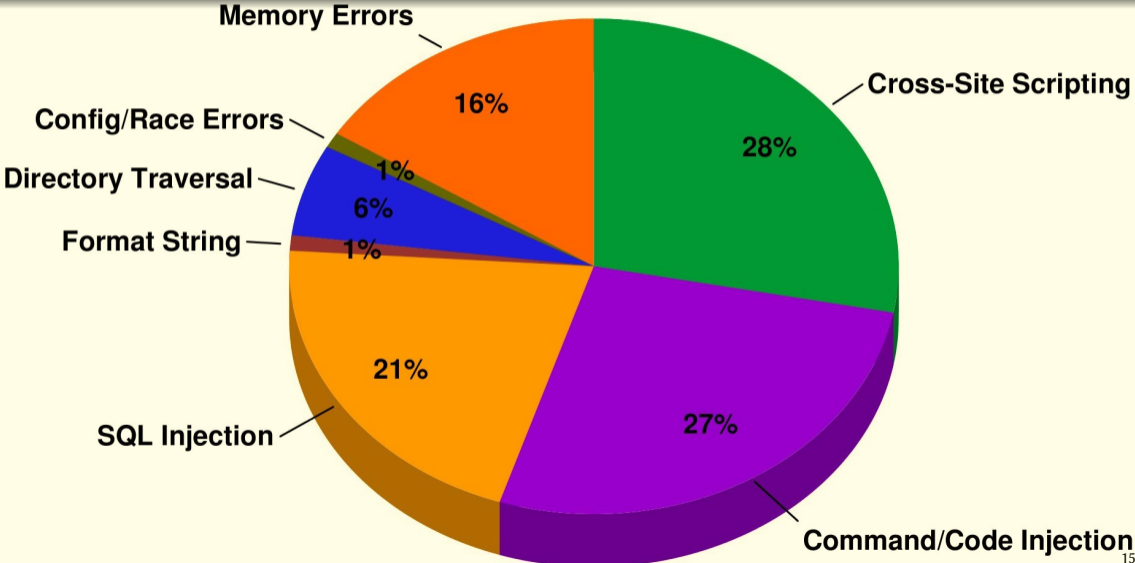
Directory traversal

- Attacker-provided path names contain directory traversal strings (e.g., "../")
- May be disguised by various encodings
 - Example:

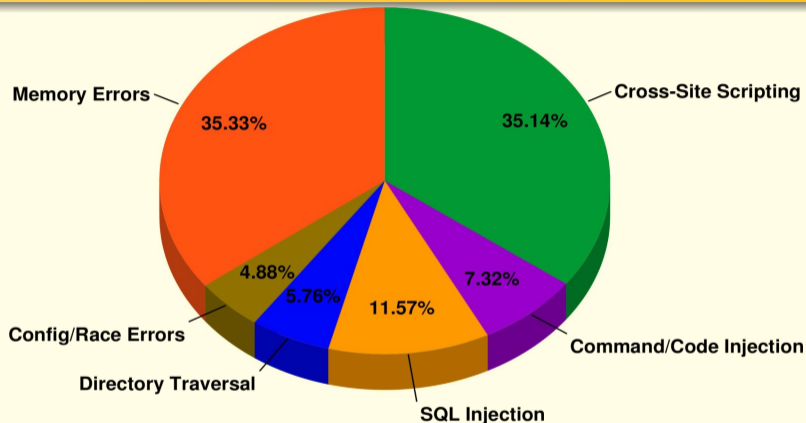
```
void check_access(char *file) {  
    if ((strstr(file, "/cgi-bin/") == file) &&  
        (strstr(file, "../") == NULL)) {  
        char *f = url_decode(file);  
        /* allow access to f ... */  
    }  
}
```

- Attacker-provided file: `/cgi-bin/%2e%2e/bin/sh`

Distribution of vulnerabilities: CVE 2006



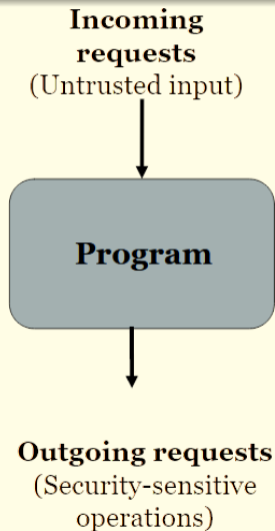
Distribution of vulnerabilities: CVE 2012



Based on CVE reports from 2012. About half of the reports correspond to specific vulnerabilities included in this chart, the rest refer to broad classes such as “logic errors” and “weak authentication.”

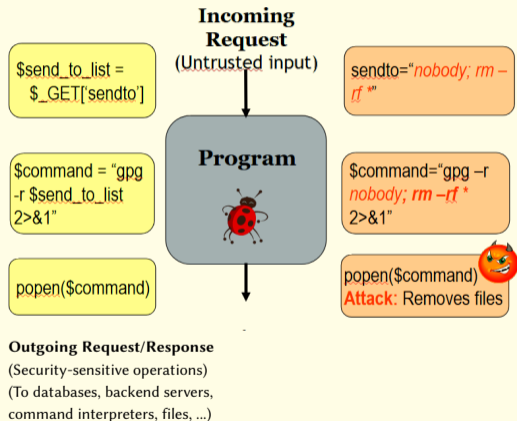
A Unified View of Attacks

- Target: program mediating access to protected resources/services
- Attack: use maliciously crafted input to exert unintended control over protected resource operations
- Resource/service access uses:
 - Well-defined APIs to access
 - OS resources
 - Command interpreters
 - Database servers
 - Transaction servers
 - ...
 - Internal interfaces
 - Data structures and functions within program meant for internal use

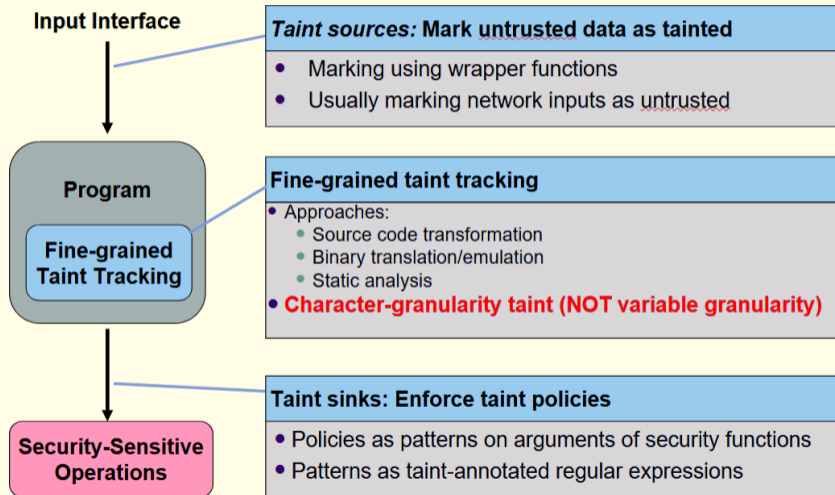


Example: SquirrelMail Command Injection

- Attack: use maliciously crafted input to **exert unintended control** over output operations
- Detect “**exertion of control**”
 - Based on taint: degree to which output depends on input
- Detect if control is intended:
 - Requires policies
 - Application-independent policies are preferable



Taint-Enhanced Policy Enforcement



Instrumentation for Taint Tracking

- Fine-grained taint-tracking
 - track if each byte of memory is tainted
- Bit array **tagmap** to store taint tags of every memory byte
- **Tag(a)**: Taint bits in **tagmap** for memory bytes at address **a**

$x = y + z; \Rightarrow \text{Tag}(\&x) = \text{Tag}(\&y) \parallel \text{Tag}(\&z);$

$x = *p; \Rightarrow \text{Tag}(\&x) = \text{Tag}(p);$

Enabling Fine-Grained Taint Tracking

- Source code transformation for C programs to track information flow at runtime
 - Accurate tracking of taint information at byte granularity
- For interpreted languages, transform the interpreter
 - If interpreter is written in C, it can be transformed as above
- Idea
 - Runtime representation of taint information
 - Use bit array `tagmap` to store taint tags for each byte of memory
 - `Tag(a)`: representing taint bits of bytes at address `a` in `tagmap`
 - Update **`tagmap`** for each assignment

Transformation: Taint for Expressions

E	$T(E)$	Comment
c	0	Constants are untainted
v	$\text{tag}(\&v, \text{sizeof}(v))$	$\text{tag}(a, n)$ refers to n bits starting at $\text{tagmap}[a]$
$\&E$	0	An address is always untainted
$*E$	$\text{tag}(E, \text{sizeof}(*E))$	
$(\text{cast}) E$	$T(E)$	Type casts don't change taint.
$\text{op}(E)$	$T(E)$ 0	for arithmetic/bit op otherwise
$E_1 \text{ op } E_2$	$T(E_1) \parallel T(E_2)$ 0	for arithmetic/bit op otherwise

Transformation: Statements

S	Trans(S)
$v = E$	$v = E;$ $\text{tag}(\&v, \text{sizeof}(v)) = T(E);$
$S_1; S_2$	$\text{Trans}(S_1); \text{Trans}(S_2)$
$\text{if } (E) S_1 \text{ else } S_2$	$\text{if } (E) \text{Trans}(S_1)$ $\text{else } \text{Trans}(S_2)$
$\text{while } (E) S$	$\text{while } (E) \text{Trans}(S)$
$\text{return } E$	$\text{return } (E, T(E))$
$f(a)\{S\}$	$f(a, ta)\{$ $\text{tag}(\&a, \text{sizeof}(a)) = ta; \text{Trans}(S)\}$
$v = f(E)$	$(v, \text{tag}(\&v, \text{sizeof}(v))) = f(E, T(E))$
$v = (*f)(E)$	$(v, \text{tag}(\&v, \text{sizeof}(v))) = (*f)(E, T(E))$

Implicit flows

- (Positive) control dependence

- Example: decoding using `if-then-else/switch`

```
if (x == '+') y = ' ';
```

- Negative control dependence `y = 1;`

```
if (x == 0)
```

```
    y = 0
```

- If `x` is tainted, but equals 1, then is `y` tainted at the end?

Implicit flows (Contd.)

- Operations involving tainted pointers

```
char transtab[256];
```

```
...
```

```
x = transtab[p]
```

- If `p` is tainted, is `x` tainted?
- What about the following case:

```
*p = 'a'
```

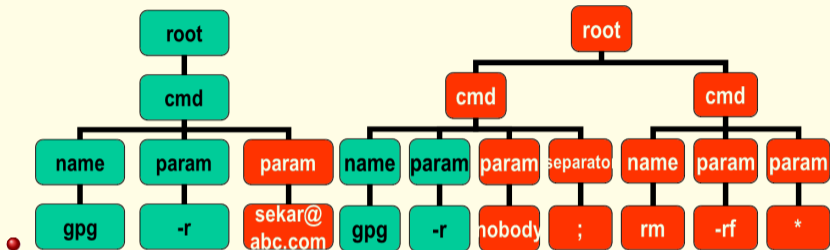
- Or the case:

```
x = hash_table_lookup(p)
```

Issues in Taint-tracking Instrumentation

- Efficiency
 - Almost every statement is instrumented
- Accuracy
 - Implicit flows
 - Full implicit flow support leads to far too many false positive
 - It is necessary to be very selective in terms of which implicit flows are taken into account.
 - Malicious code can disguise all flows in implicit flows, making it infeasible to do accurate taint-tracking
 - Untransformed libraries

Application-independent policies



- Lexical confinement
 - Ensure that tainted data does not cross a word boundary
 - For binary data, can interpret struct fields as words
 - Or more coarsely, activation records or heap blocks
- Syntactic confinement (more relaxed)
 - Tainted data should not begin in the middle of one subtree of the parse tree and “overflow” out of it

Symlink attacks

- Do not assume that symlinks are trustworthy:
 - Example 1
 - Application A creates a file for writing in /tmp. It assumes that since the file name is unusual, or because it encodes A's name or pid, there is no need to check if the file is already present
 - Attacker creates a symlink with same name that points to an important file F. When root runs A, F will be overwritten.
 - Example 2
 - User A runs an application that creates a file in /tmp/x and then later updates it.
 - User B attacks this application by removing /tmp/x and then creating a symlink named /tmp/x that points to an important file F.
- Hard links and file/directory renames can also be used to carry out some of these attacks, but they are difficult because there are more restrictions on them.

Race conditions

- Time-of-check-to-time-of-use (TOCTTOU) attacks
 - Often arise when an application tries to protect itself against name-based attacks
- Example
 - A setuid application permits a non-root user to specify the name of an output file, say, for logging
 - It checks if the real user has permission to write this file, usually using the access system call
 - Attacker modifies the file between access and open
 - Checks OK, but the attack succeeds!

Race condition examples

- access/open
- chmod/chown
- Directory renames
 - Root invokes `rm -r` on `/tmp/*` to clean up `/tmp`
 - Attacker creates a directory `/tmp/a` and then another directory `/tmp/a/b`
 - `rm` may (1) `cd` into `/tmp/a/b`, remove all files in it, (2) `cd` into `“..”`, (3) continue to remove files in `/tmp/a`, (4) `cd` `“..”` and (5) continue to remove files in `/tmp`
 - Attacker moves `/tmp/a/b` to `/tmp` between (1) and (3), causing files in `/` to be removed in step (5).

Succeeding in Races ...

- It may seem that it would be hard for the attacker to succeed, but he can mount “algorithmic complexity attacks”
 - Make a normally fast operation take very long
 - Example: Instead of creating a file `/tmp/a`, make it point to a symlink which in turn points to a symlink and so on. Access operation, which needs to resolve this sequence of symlinks will take very long. Can further slow it down by creating deep directory trees.
 - As a result, races can succeed with near 100% probability!

Avoiding filename related pitfalls

- When creating new files, call open with appropriate flags to ensure creation of new file
 - On UNIX, `O_CREAT` and `O_EXCL` flags
- Use OS-provided functions to create temp files
 - On UNIX, use `mkstemp` or `tmpfile`, not `tmpnam`
- Use most restrictive permission applicable
 - Always restrict writes to owners, and if possible, reads too.
 - *If possible, first create a directory that is accessible only to the owner, and operate within this directory*
- Configure shared directory permissions correctly
 - Use the sticky bit

Common Software Weaknesses

- Input validation
 - Injection vulnerabilities
 - Cross-site scripting, SQL/command injection, code/script injection, format-string, path-traversal, open redirect, ...
 - Buffer overflows
 - integer overflows, incorrect buffer size or bounds calculation
 - Many other application-specific effects of untrusted input
- Failure to recognize or enforce trust boundaries
 - Calling function that trust their inputs with untrusted data
 - Including code without understanding its dependencies
 - Relying on form data or cookies in a web application

Common Software Weaknesses

- Missing security operation
 - Authentication: missing, weak, or using hard-coded credentials
 - Authorization: missing checks
 - Cross-site request forgery
 - Failure to encrypt, hash, use salt, ...
- Use of weak security primitives
 - Weak random numbers, encryption, hash algorithms, ...
- Information leakage
 - Error messages that reveal too much information
 - Software version, source code fragments, database table names or errors, ...
 - Timing channels

Common Software Weaknesses

- Execution with unnecessary privileges
 - Executing code with admin privileges
 - Incorrect (or missing) permission settings
- Error/exception-handling code
 - Failure to check error codes, e.g., open, malloc, ...
 - Failure to test error/exception-handling code
- Race conditions

CVEs and CWEs

- CVE (Common Weakness Enumeration) is used to catalog and document specific vulnerabilities
 - Identifies the software involved, conditions for exploitation, mitigations, severity, etc.
- CWE (Common Weakness Enumeration) is broader categorization of vulnerability types
- CWE Top-25 is a good point to start
 - You are expected to be familiar with the vulnerabilities in this list — read the list and understand what each vulnerability means

Secure Coding Practices

- The goal of this course is to expose you to a range of vulnerabilities and exploits, so you can learn how to build secure systems and develop secure code
- But we don't necessarily provide a “cook book”
 - The hope is that you will learn more from understanding the examples in depth than reading a long laundry list
- Nevertheless, several good sources are available on the Internet that discuss secure coding practices
 - SEI CERT top 10 secure coding practices
 - CERT Secure coding standards for C, C++, and Java
 - OWASP Secure coding principles

Principles of Secure System Design

- [Saltzer and Shroeder 1975]
- Principles of
 - Economy of mechanism (simplicity => assurance)
 - Fail-safe defaults (default deny)
 - Complete mediation (look out for ways in which an access control mechanism may be bypassed)
 - Open design (no security by obscurity)
 - Separation of privilege (similar to separation of duty)
 - Least privilege
 - Least common mechanism (avoid unnecessary sharing)
 - Psychological acceptability (onerous security requirements will be actively subverted by users)

Principles of Secure System Design

- Two principles mentioned, but not recommended in [Saltzer and Schroeder 1975]
 - Work factor: how much effort will it take to break a mechanisms, versus potential gain for the attacker
 - Difficult to estimate cost
 - Sometimes, difficult to estimate gain
 - Compromise recording (maintain adequate audit trail)
 - Difficult to ensure integrity of audit records maintained on a protected system
 - These records can be compromised if stored on protected system
 - Can work if audit trail can be protected, e.g., off-site storage, tamper-proof storage systems

Vulnerabilities Vs Malicious Code

- These two pose very different threats
 - With vulnerable code, you have a relatively weak adversary: one that is constrained to exploiting an existing vulnerability, but has no way of controlling it.
 - So, relatively weak defenses such as randomization can be attempted.
 - With malicious code, you have a strong adversary
 - Can modify code to evade specific defenses
 - You cannot make assumptions such as the absence of intentionally introduced errors, obfuscation, etc.