UNIVERSITY OF WESTMINSTER

Software and Web Security

6COSC019W- Cyber Security

Dr Ayman El Hajjar

March 05, 2024

School of Computer Science and Engineering
University of Westminster

Outline

1. Web Application Attacks

2. Application Exploitation

3. Countermeasures

4. Software Development Security

- Many computer security vulnerabilities result from poor programming practices
- The OWASP Top 10 is a standard awareness document for developers and web application security. It represents a broad consensus about the most critical security risks to web applications.
  - Broken Access Control
  - Cryptographic Failures
  - Injection
  - Insecure Design
  - Security Misconfiguration
  - Vulnerable and Outdated Components
  - Identification and Authentication Failures
  - Software and Data Integrity Failures
  - Security Logging and Monitoring Failures
  - Server-Side Request Forgery (SSRF)

SECURITY FLAWS

❑ Critical Web application security flaws include five related to insecure software code

1. Handling Program handling
2. Buffer overflow
3. Injection flaws
4. Cross-site scripting
5. Improper error handling

❑ These flaws occur as a consequence of insufficient checking and validation of data and error codes in programs

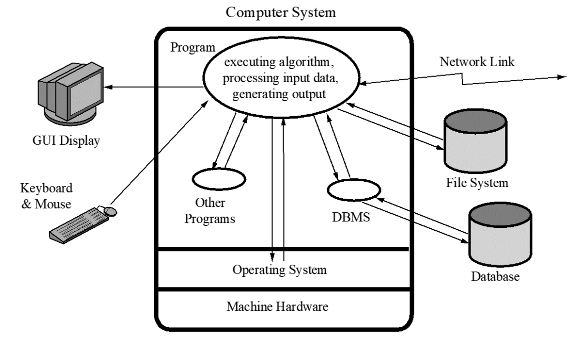
❑ Awareness of these issues is a critical initial step in writing more secure program code

❑ Emphasis should be placed on the need for software developers to address these known areas of concern

CWE/SANS TOP 25 MOST DANGEROUS SOFTWARE ERRORS

- 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 defences

ABSTRACT VIEW OF PROGRAM



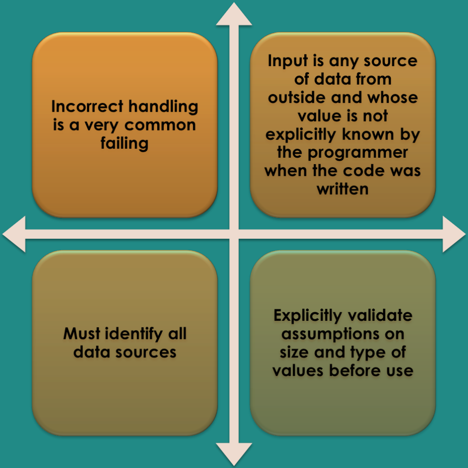
SOFTWARE SECURITY, QUALITY AND RELIABILITY

- ❑ 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

Web Application Attacks

Handling Program Input

❏ Unvalidated input, one of the most common failings in Web application 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

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

SQL Injection Attacks (SQLi)

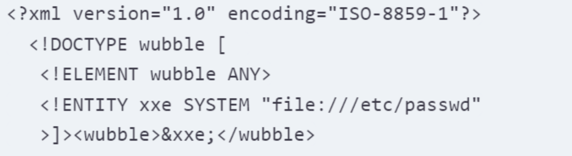
- ❑ One of the most prevalent and dangerous network-based security threats
- ❑ Designed to exploit the nature of Web application pages
- ❑ An SQL injection attack consists of insertion or “injection” of a SQL query via the input data from the client to the application
- ❑ Most common attack goal is bulk extraction of data
- ❑ A successful SQL injection exploit can:
  - ❐ Read sensitive data from the database
  - ❐ Modify database data (Insert/Update/Delete)
  - ❐ Execute administration operations on the database (such as shutdown the DBMS)
  - ❐ Recover the content of a given file present on the DBMS file system
  - ❐ and in some cases issue commands to the operating system.

XML External Entity Processing

❑ A common way to pass data back and forth between a client and a server is to use XML to structure the data and transmit that XML.

❑ An XML entity injection attack comes from code on the web server accepting data that comes from the client without doing any data validation.

❑ We can even tamper with an existing XML page and parse different commands through the XML page.



```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE wubble [
   <!ELEMENT wubble ANY>
   <!ENTITY xxe SYSTEM "file:///etc/passwd"
   >]><wubble>&xxe;</wubble>
```

Figure 1: XML External Entity Processing Example

Cross Site Scripting (XSS) Attacks

❑ A cross-site scripting (XSS) attack is one that uses the web server to attack the client side.

❑ This injects a code fragment from a scripting language into an input field to have that code executed within the browser of a user visiting a site. For example <script></script> block

❑ There are three types of cross-site scripting attack. The difference is whether the script is stored somewhere or not.

- ❒ **Persistent cross-site scripting**. Stored on the server and displayed for any user visiting a page
- ❒ **Reflected cross-site scripting**. : The script isn't stored. Instead, it is included in a URL as a parameter you would send to a victim.
- ❒ **DOM-based XSS attack**: Document Object Model (DOM)-based XSS attack allow us to call for objects through the scrip which should result in the object being executed.

Injection Technique: SQLi

❏ 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 “- -”

↓

❏ 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

TYPE OF SQL INJECTIONS

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 behaviour of the Website/database server.

SQLI ATTACK EXAMPLE: LOGIN AUTHENTICATION QUERY

- Standard query to authenticate users:

  select * from users where user='$usern' AND pwd='$password'

- Classic SQL injection attacks

  Server side code sets variables *usernameand*passwd from user input to web form

  Variables passed to SQL query

  select * from users where user='$username' AND pwd='$password'

- Special strings can be entered by attacker

  select * from users where user='M' OR '1=1' AND pwd='M' OR '1=1'

- Result: access obtained without password

COMMAND INJECTION ATTACK

❑ Similar to an XML external entity injection attack.

❑ The application takes a value from the user and passes it to a system function or an evaluate function.

❑ Focus on the operating system

❒ Pass the parameters to the operating system to handle.

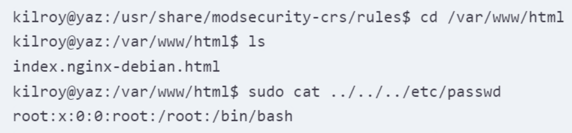
❑ **Consequences**: If there is no input validation, you can execute any operating system command into the input field

❑ For example: If I enter ping -c 5 192.168.56.111 && cat /etc/passwd, the result will be the ping and the contents of the passwd file.

❒ Try this in next week lab: Lab 5- Finding and Exploiting Web Vulnerabilities

FILE TRAVERSAL

- ❑ File traversal is a way to get out of what the web server wanted you to originally see, and be able to see more.
- ❑ For example: The default web-server public folder for Apache server on Linux is /**var**/**www**/**html**
- ❑ If we visit the website of this web-server. the server will point us to the /**var**/**www**/**html** , usually an index.html page (or whatever language the site is written in)
- ❑ File Traversal is the ability to browse the web server and see files outside the contents of /**var**/**www**/**html** , for example root folder of the web server
  - ❑ The web-server public folder for Apache server on our OWASP VM is /var/www/



```
kilroy@yaz:/usr/share/modsecurity-crs/rules$ cd /var/www/html
kilroy@yaz:/var/www/html$ ls
index.nginx-debian.html
kilroy@yaz:/var/www/html$ sudo cat ../../../etc/passwd
root:x:0:0:root:/root:/bin/bash
```

Application Exploitation

BUFFER OVERFLOW ATTACK

- ❏ A very common attack mechanism
- ❏ 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

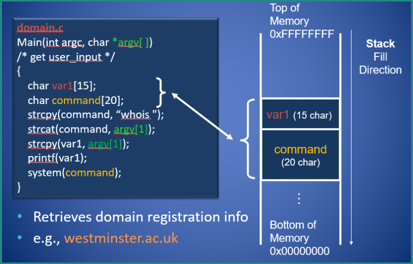
Overflow attack types

- ❑ 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 attacker exploits an unchecked buffer to perform a buffer overflow attack

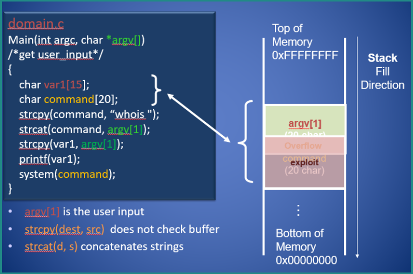
❑ The ultimate goal for the attacker is getting a shell that allows to execute arbitrary commands with high privileges



```
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

Figure 3: Memory contents Before the malicious input



```
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

Figure 4: Memory contents after the malicious input, causing the Buffer Overflow

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

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

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

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 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

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 value immediately 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

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

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

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

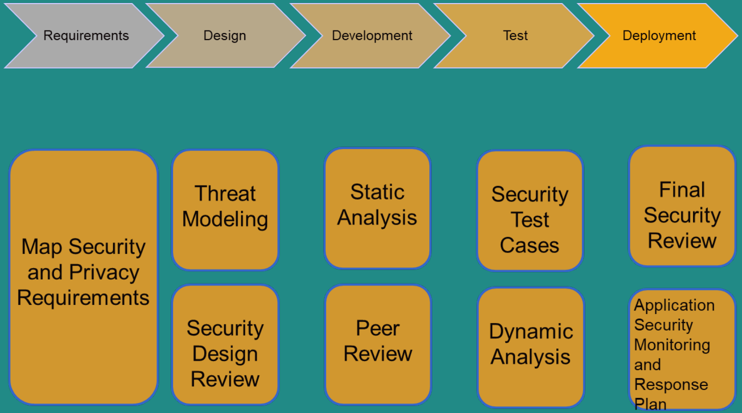
❑ Phases of SDLC

- ❒ 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

SECURE SOFTWARE DEVELOPMENT LIFE CYCLE



Secure Software Development Life Cycle

Requirements:

- ❑ Map security and privacy requirements
  - ❒ Business system analysis should be familiar with organisational security policies and standards such as organisation privacy policy and regulatory requirements.

Development

- ❑ Threat modelling
  - ❒ Used to determine the technical security posture of the application being developed
- ❑Design reviews
  - ❒ Carried out by a security subject matter expert and typically iterative in nature

Secure Software Development Life Cycle

Development

- ❑ Development-related vulnerabilities
  - ❒ Static analysis: Automation find issues with source code
  - ❒ Peer review: Developers review each others code and provide feedback

Testing

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

Deployment

- ❒ Final security review
  - ❒ Create application security monitoring and response plan
  - ❑ Security training

REFERENCES

❑ The lecture notes and contents were compiled from my own notes and from various sources.

❑ Figures and tables are from the recommended books

❑ **The lecture notes are very detailed. If you attend the lecture, you should be able to understand the topics.**

❑ **You can use any of the recommended readings! You do not need to read all the chapters!**

❑ **Recommended Readings note:** Focus on what was covered in the class.

- ❒ Chapter 12- Attack and Defence, CEH v11 Certified Ethical Hacker Study Guide
- ❒ SQL Injection on Owasp site Link
- ❒ Chapter 8, Malicious Software and Attack Vectors, Fundamentals of Information Systems Security
- ❒ Chapter 15, 16 & 17, CyBOK, The Cyber Security Body of Knowledge