Based on Common Weakness Enumeration (CWE) declared in the https://cwe.mitre.org/, identify the compliance ID and explain on defensive coding implementation for each error type below.

i. SQL Injection

ii. Cross-site Scripting

iii. OS Command Injection

iv. Classic Buffer Overflow

v. Missing Authentication for Critical Function

SQL Injection

- A code injection technique that manipulates SQL queries by inserting malicious SQL statement
- Attacker inserts malicious SQL code into an application input field or URL parameter to bypass authentication or manipulate database data.
- Use parameterized queries (ORM)
- Implement input validation
- Principle of least privilege for database accounts

XSS

- Allows attackers to inject malicious scripts into web pages viewed by other users
- Attacker inserts malicious JavaScript into web application, which is then executed in victim browser
- Stored XSS, Reflected XSS
- Implement output encoding
- Use Content Security Policy
- Validate and sanitize user input

OS Command Injection

- Allows execution of arbitrary command through a vulnerable app
- Attackers inject system commands into application inputs that are passed to shell
- Avoid using system commands when possible
- Use whitelists for permitted commands
- Validate and sanitize user input

Classic Buffer Overflow

- Occurs when a program writes more data to a buffer than it can hold
- Attackers input more data than expected, overwriting adjacent memory and potentially executing arbitrary code
- Use safe string handling functions
- Implement bound checking
- Enable compiler auto bound assigning

Missing Authentication for Critical Function

- Occurs when application fails to require authentication for critical function
- Attackers can access sensitive functions or data without proper authentication
- Implement strong authentication and authorization
- Least privilege
- MFA, strong password policies

Cross Site Request Forgery

- Tricks the victim into submitting malicious request to a web app where they are authenticated
- Attackers create malicious website that send unauthorized requests to trusted sites using victim's credentials
- Generate unique anti CSRF tokens for each user sessions, include in forms
- Use SameSite cookie attributes, prevent cookies from being sent in cross site requests
- Verify the CORS and referrer headers

SSRF

- Allows an attacker to induce the server-side application to make requests to an unintended location
- Attackers manipulate the server to send requests to internal or external systems, potentially bypassing security controls
- Disable unused URL schemas
- Implement whitelist of allowed URLs/IP address
- Use a separate, unprivileged service for making external requests

SSDLC – Analyse, Planning (Study source of threat), Development, Testing, Distribution

Fail secure, economy mechanism principle, complete mediation, psychology acceptability

You are given TWO (2) different codes as below. Analyse each piece of the codes given and explain what the bad and good implementation in the codes are if any. The explanation may include the vulnerability of the code, good coding implementation in preventing certain kind of attacks and any discussion that relevant to secure coding.

(20 marks)

Code 1:

```java
Public void upload(HttpServletRequest request) throws ServletException {

        MultipartHttpServletRequest mRequest = (MultipartHttpServletRequest) request;
        String next = (String) mRequest.getFileNames().next();
        MultipartFile file = mRequest.getFile(next);
         if (file = = null) return;

         if (fileName != null) {
         if (fileName.endsWith(".docx") || fileName.endsWith(".pptx")
         || fileName.endsWith(".pdf") || fileName.endsWith(".xlsx")){

        /* file upload routine*/
        } else
        throw new ServletExeption("error");
}
```

Code 2:

```java
Public class FibonacciNumbers{
        public int fibonacci(int n){
                return fibonacci(n – 1) + fibonacci(n – 2);
        }
        public static void main(String[] args){
                FibonacciNumbers f = new FibonacciNumbers();
                int x = f.fibonacci(10);
                System.out.println(x);
        }
}
```

Good implementations:

- The code checks if the file is null before proceeding, which prevents null pointer exceptions.
- It validates the file extension, allowing only specific file types (.docx, .pptx, .pdf, .xlsx). This helps prevent uploading of potentially malicious file types.

Areas for improvement:

- The file extension check is case-sensitive. A malicious user could bypass it by using uppercase extensions (e.g., .PDF).
- The code doesn't validate the file content, only the extension. A malicious file could be renamed to have an allowed extension.
- The error handling is overly generic, throwing a ServletException with just "error" as the message. More specific error messages would be helpful for debugging and user feedback.
- There's no size limit check on the uploaded file, which could lead to denial-of-service attacks through very large file uploads.
- The code doesn't sanitize the file name, which could potentially lead to path traversal attacks if the filename is used directly in file system operations.

Security recommendations:

- Implement case-insensitive extension checking.
- Add content-type verification in addition to extension checking.
- Implement file size limits.
- Sanitize and validate all user inputs, including file names.
- Use more specific error handling and logging.
- Consider using a library for secure file upload handling instead of implementing it from scratch**.**

Good implementations:

- The code is concise and follows a simple recursive approach to calculate Fibonacci numbers.

Areas for improvement:

- Lack of base cases: The fibonacci method doesn't handle base cases (n = 0 and n = 1), which will lead to infinite recursion and a StackOverflowError.
- Inefficient algorithm: The recursive implementation has exponential time complexity O(2^n), making it extremely slow for larger values of n.
- No input validation: The code doesn't check if the input n is non-negative, which could lead to unexpected behavior or errors for negative inputs.

- Potential integer overflow: For larger Fibonacci numbers, the int data type may overflow, leading to incorrect results.
- No exception handling: The code doesn't handle potential exceptions that might occur during execution.

Security and performance recommendations:

- Add base cases to prevent infinite recursion:
- Consider using an iterative approach or dynamic programming for better performance.
- Implement input validation to ensure n is non-negative.
- Use long instead of int to handle larger Fibonacci numbers or consider using BigInteger for arbitrary precision.
- Add exception handling to manage potential errors gracefully.
- For very large n values, consider implementing a more efficient algorithm like matrix exponentiation.

Adversary behavioral identification involves the identification of the common methods or techniques followed by an adversary to launch attacks on or to penetrate an organization's network. Discuss on any **4 (FOUR)** of adversary behaviors.

Reconnaissance

- Is the first step in adversary playbook, gather information about the organization, including network infra, employee information, potential vulnerabilities
- Can involve techniques such as port scanning, social engineering, and OSINT gathering
- Security team can implement measures to limit publicly available information, implement threat hunting to detect early signs of probing activities

Initial Access

- Adversaries attempt to gain access into the organization network
- Can involve techniques such as phishing emails, exploiting unpatched vulnerabilities, weak or stolen credentials
- May also perform sophisticated techniques like supply chain or zero-day exploits
- Access controls, educate employees, authentication mechanisms

Lateral Movements

- To expand their access to high privileges target
- Can involve techniques such as privilege escalation, credential dumping, exploit trust relationship between system
- May use LOLBins or custom malware to avoid detection while moving through network
- Implement network segmentation, deploy EDR to monitor for unusual activities, MFA

Exfiltration

- End goal of adversaries
- Involves unauthorized transfer of sensitive information out of the network.
- Can involve techniques such as encrypted tunnels, steganography, or cloud services.
- May stage the data before exfiltration, compress it to avoid detection, exfiltrate it in small chunk overtime
- Implement DLP tools, Firewall, implement strict read/write access

Analyse the code below:

```
#include <stdio.h>
main() {
  char *name;
  char *dangerous_system_command;
  name = (char *) malloc(10);
  dangerous_system_command = (char *)
  malloc(128);
  printf("Address of name is %d\n", name);
  printf("Address of command is %d\n",
  dangerous_system_command);
  sprintf(dangerous_system_command, "echo
  %s", "Hello world!");
  printf("What's your name?");
  gets(name);
  system(dangerous_system_command);
}
```

Based on the code above identify the vulnerability that exist the the codes above.

Suggest any amendment required in the codes to minimize the vulnerabilities that exist in the codes.

Buffer Overflow: The 'name' variable is allocated only 10 bytes, but the gets() function is used to read user input without any bounds checking. This can lead to a buffer overflow if the user enters more than 10 characters.

Command Injection: The code uses the system() function to execute a command stored in dangerous_system_command. This variable is populated using sprintf(), which doesn't perform any input sanitization, making it vulnerable to command injection attacks.

Memory Leaks: The code allocates memory using malloc() but never frees it, leading to memory leaks.

Use of Dangerous Functions: The gets() function is considered dangerous and deprecated due to its lack of bounds checking.

To minimize these vulnerabilities, there are several secure code practices to follow including

1. Replace gets() with a safer alternative like fgets(). For example: fgets(name, 10, stdin);
2. Avoid using system() for command execution. If system commands are necessary, use execve() with proper argument sanitization.

3. Implement proper input validation and sanitization before using user input in any system command. For example: regex implementation to remove special characters and whitespaces.
4. Use strncpy() or snprintf() instead of sprintf() to prevent buffer overflows. For example: snprintf(dangerous_system_command, 128, "echo %s", "Hello world!");
5. Free allocated memory to prevent memory leaks. For example:

```
free(name);
free(dangerous_system_command);
```

Discuss any good or bad things about the code and interface design below:
You may include any suggestion for improvement.

```
Option Explicit
Dim objNetwork, strDrive, strRemotePath

strDrive = "J:"
strRemotePath = "\\FinServer\Software"

On Error Resume Next

Set objNetwork = CreateObject("WScript.Network")
objNetwork.MapNetworkDrive strDrive, strRemotePath

Wscript.Quit
```

Good Aspects:

Use of Option Explicit:

The Option Explicit statement is included, which is a good practice as it forces the declaration of all variables. This helps prevent errors caused by typos or incorrect variable names.

Modular and Clear:

The code is modular and clear, with variables like strDrive and strRemotePath being declared and used in a straightforward manner. This makes the code more readable and maintainable.

Bad Aspects:

Use of On Error Resume Next:

The line On Error Resume Next is used to suppress any runtime errors. While this might prevent the script from crashing, it also hides all errors, making it difficult to debug and understand why certain parts of the script may not be working correctly.

Suggestion: Instead of using On Error Resume Next, implement proper error handling by using On Error GoTo 0 after the critical section, and log or handle specific errors to understand the issue better.

Lack of Error Handling:

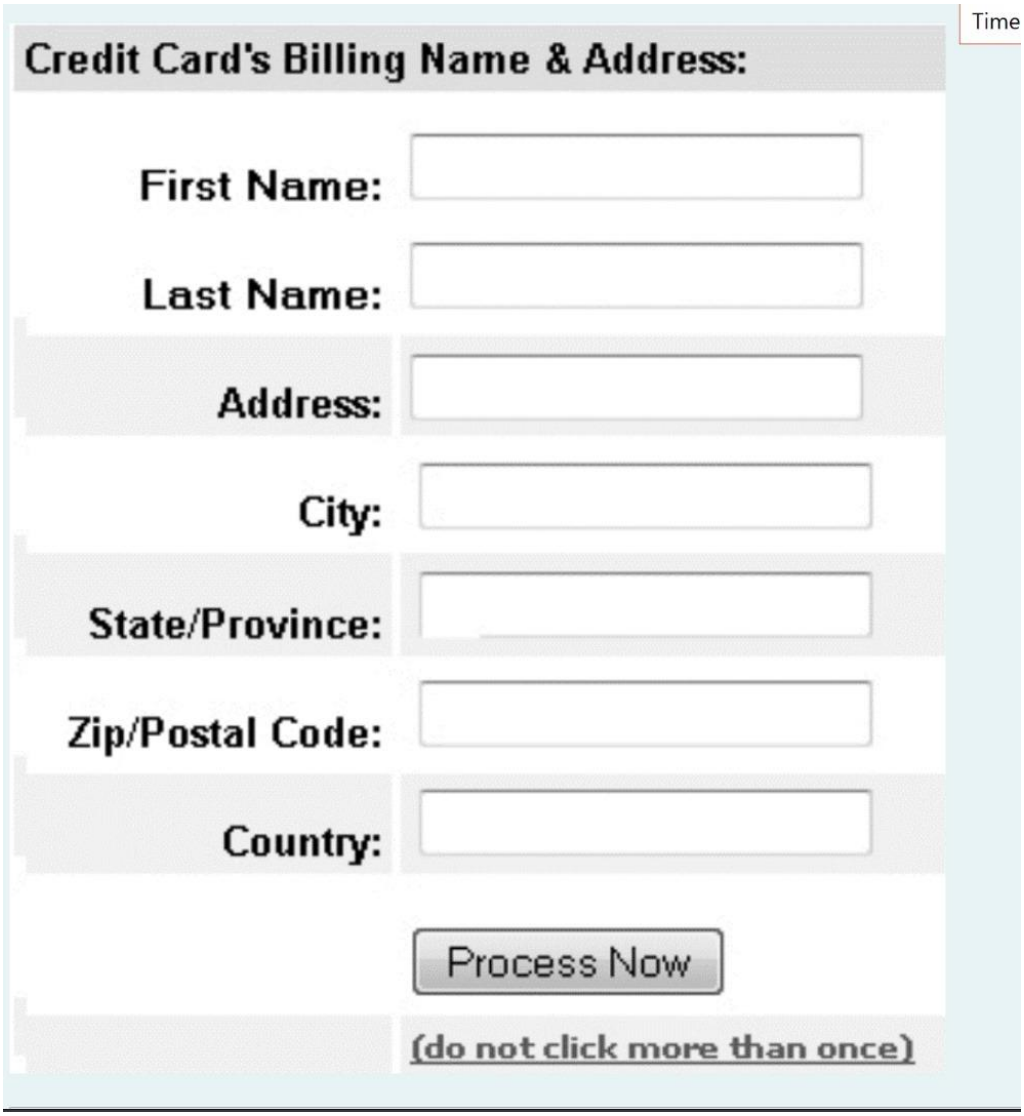
There is no error checking after the network drive mapping operation (MapNetworkDrive). If the operation fails (e.g., due to an incorrect path, lack of permissions, or network issues), the script will not provide any feedback to the user or log the error.

Suggestion: Implement error handling after the network drive mapping attempt. For example, check if the drive was successfully mapped and provide an error message or log the error if it fails.

Hardcoded Values:

The values for strDrive and strRemotePath are hardcoded. This limits the flexibility of the script, as it can only map a specific drive to a specific path.

Suggestion: Consider allowing these values to be passed as arguments or read from a configuration file, making the script more reusable and adaptable to different environments.



Good Aspects:

Clear Labels:

Each input field is clearly labeled, making it easy for users to understand what information is required. For example, fields like "First Name," "Last Name," "Address," etc., are straightforward.

Logical Layout:

The form fields are organized logically, with fields grouped in a sequence that aligns with typical address information entry. This makes the form easier to fill out.

Call to Action:

The "Process Now" button provides a clear call to action, indicating what will happen when the user submits the form.

Bad Aspects:

Lack of Field Validation Indicators:

There are no indicators or hints suggesting which fields are required and what the expected format is. For instance, the "Zip/Postal Code" field may require a specific format (e.g., numeric or alphanumeric), but there is no guidance provided.

No Real-Time Feedback:

The form does not provide real-time validation feedback. Users might not know if they've made a mistake until they submit the form, which can be frustrating and lead to errors.

Non-Distinguishable Error Handling:

There is a note under the "Process Now" button saying, "do not click more than once." This suggests that the form might not handle multiple submissions gracefully, which could cause issues like duplicate submissions. Modern forms should handle this internally by disabling the button or providing feedback when the form is being processed.

Non-Responsive Design:

The form appears to be designed for a fixed-width layout, which might not be optimal for mobile devices or smaller screens. This could result in a poor user experience for mobile users.

No Security Indicators:

Since the form deals with sensitive information (credit card billing details), there should be some indication that the form is secure, such as an SSL certificate icon or a note about data encryption.