Practice 2

Secure Development Exercise – Secure Code

Name: Madizhan Islambek

Email: p42isdam@uco.es

Professor: Juan Antonio Romero Castillo

Answers to Questions

Q1. What: A self-replicating program that exploited vulnerabilities in UNIX systems.

Where: Originated at MIT.

How: Exploited flaws in systems like weak passwords and buffer overflow vulnerabilities.

When: November 2, 1988.

Why: Designed as an experiment but caused unintentional damage, infecting approximately 10%.

Q2. When a progam lets a function to write more data than a buffer can hold a buffer Overflow Attack can happen which leads to memory ovverwrites. A hacker can use that space to execute an arbitrary code. Example:

```
void vulnerable_function() {

   char buffer[10];

   strcpy(buffer, "This is a long input that overflows");

}
```

Q3. What is a Double Free Attack?

A Double Free Attack occurs when memory is freed more than once, potentially corrupting the program's memory management. Example:

```
int main() {

   int *ptr = (int *)malloc(sizeof(int));

   free(ptr);

   free(ptr); // Double free

   return 0;}
```

simplest way to prevent this is to use NULL:

```
int main() {
   int *ptr = (int *)malloc(sizeof(int));
   free(ptr)=NULL;
   free(ptr);
   return 0;}
```

Another way to prevent Double Free Attack is to use Valgrind or zlib library.

Q4. Analysis of C Code Examples

a) The problem:

This code checks if a+1 is greater than a, which is always true for valid integers. However, it may fail for edge cases like integer overflow.

b) The problem:

Using *i after free(i) causes undefined behavior. The memory has been deallocated, and writing to it can corrupt other parts of the program.

c)The problem:

If buf2 contains more data than buf, this code will cause a buffer overflow in buf. The condition does not verify the bounds of buf.

d) The problem:

If n is not properly checked, this code can cause memory allocation issues, leading to potential overflows or allocation failures.

Q5. Secure Coding Practices

- **Don’t Ignore Compiler Warnings**

Compiler warnings indicate potential issues in the code. Ignoring them can lead to vulnerabilities and bugs.

- **Don't Write Complex Code**

Simple code is easier to debug, maintain, and audit for security vulnerabilities. Let's see that in example:

Complex code example:

```
int factorial(int n) {
    if (n < 0) {
        return -1; // Error for negative input
    }
    int result = 1;
    for (int i = 1; i <= n; result *= i++);
    return result;
}
```

Simpler code example:

```
int factorial(int n) {
    if (n < 0) {
        printf("Error: Negative numbers don't have a factorial.\n");
        return -1; // Error code for invalid input
    }

    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i; // Clearly separated logic
    }
```

return result;

}

Why we need to do this?

Readability: The logic is straightforward, with separate operations for clarity.

Error Handling: The error message for invalid input makes debugging easier.

Maintainability: Adjusting the loop or adding new functionality (e.g., logging) is easier due to clear structure.

Auditing for Security: With clear separation of conditions and operations, it's easier to identify potential vulnerabilities like buffer overflows or input issues.

- ### Use Enum for Error Codes

Using enum provides a clear and readable way to handle error codes. similar to handling exceptions while testing a program, using enums helps naming the errors and addressing them.

Example:

c

Кодты көшіру

```
typedef enum {
    FILE_OPEN_ERROR = 0,
    FILE_CLOSE_ERROR,
    FILE_READ_ERROR
} FILE_ERROR_LIST;
```

- ### Use Fixed-Width Data Types

Fixed-width types (e.g., uint8_t, uint16_t) ensure consistent behavior across platforms.

Potential issues prevented by Fixed_Width data types:

- Buffer Overflow
- Truncation Errors
- Portability Issues
- Network Protocol Mismatches
- Arithmetic Overflows/Underflows

Q6. Analysis of Output:

The result will depend on how iData + uiData is handled due to typecasting. The signed integer (iData) may be converted to an unsigned integer, causing unexpected behavior.

Expected Output: Likely "a+b > 6" because the signed negative iData converts to a large unsigned value during addition.