Group #7 (Trivon Paul, Minh Tram)

4/4/2024

CS-455-01 Spring 2024

Project Implementation Analysis

**Five types of vulnerabilities in the Caesar’s cipher:**

1. Buffer Overruns

buffer = (char \*)malloc(file\_size + 1); // +1 for null terminator

if (buffer == NULL) {

}

// Read the file content into the dynamically allocated memory

if (fread(buffer, 1, file\_size, file) != file\_size) {

}

// Null-terminate the buffer to make it a valid C string

buffer[file\_size] = '\0';

=> added space for the null terminator

char \*\*dirArray;

dirArray = (char \*\*)malloc(length \* sizeof(char \*));

if (dirArray == NULL) {

printf("Memory allocation failed for the array of strings.\n");

return 1;

}

// Allocate space for each string

for (int i = 0; i < length; i++) {

dirArray[i] = (char \*)malloc(MAXPATH \* sizeof(char));

if (dirArray[i] == NULL) {

printf("Memory allocation failed for string %d.\n", i);

return 1;

}

}

=> By using malloc() instead of \*dirArray[length], we prevent the possibility of a buffer overflow into this variable. If we were to use \*dirArray[length], the first file we encrypt would be fine, but the contents in char \*buffer would overflow into \*dirArray[length]. Therefore, by using malloc(), we avoid this overflow.

2. Catching Exceptions

1. File Opening : The code checks if the file can be opened successfully using 'fopen'.
2. Memory Allocation: Memory allocation using malloc is checked, and if it fails, an error message is printed, and the program returns with an error code.
3. File Reading and Writing: The code uses fread to read the file content into the dynamically allocated buffer and fwrite to write the modified content back to the file. If either of these operations fails, an error message is printed, and the program returns with an error code.
4. User Input: The code uses scanf to get user input for the shifting number. If the input is not valid (e.g., not an integer), an error message is printed, and the program returns with an error code.
5. Directory Opening: The code checks if the directory can be opened using opendir().
6. File Status: The code checks if the file status can be read using lstat().

3. Integer Overflows

if (file\_size < 0 || file\_size + 1 > SIZE\_MAX) {

fprintf(stderr, "File size exceeds the supported limit\n");

fclose(file);

return 1; // Exit with an error code

}

=> This ensures that the total size still falls within the representable range of size\_t (given by SIZE\_MAX).

This way, we prevent potential overflow issues related to dynamic memory allocation and ensure that there is enough space for the null terminator.

4. Command Injection

if ((file = fopen(argv[1], "rb")) == NULL) {

}

if ((file = fopen(argv[1], "wb")) == NULL) {

}

=> Avoid Constructing Commands with User Input:

This usages is generally safe because fopen is used to open a file, and the second argument "rb" specifies that it should be opened in binary read mode.

However, if we were to construct shell commands or system calls using user input, it could lead to command injection vulnerabilities.

mkdir(backupDir, 0777);

=> By using mkdir() instead of executing a system call using user input we avoid command injection vulnerabilities while still being able to complete the required objective which is creating a new directory if it doesn’t already exists.

5. Failure to Handle Errors Correctly

1. Opening the File:

if ((file = fopen(argv[1], "rb")) == NULL) {

perror("Error opening file for reading");

return 1; // Exit with an error code

}

=> This part correctly checks if the file can be opened and uses perror to print an error message with additional information from the operating system.

It then returns an error code.

1. Memory Allocation:

buffer = (char \*)malloc(file\_size + 1); // +1 for null terminator

if (buffer == NULL) {

perror("Memory allocation failed");

fclose(file);

return 1; // Exit with an error code

}

=> Similarly, this part checks if memory allocation was successful and prints an error message if it fails.

It then releases the resources (closes the file) and returns an error code.

1. Check Return Values:

if(directoryLength(dirName) == -1) return 1; // Exit with an error code

=>After a function call you should check to see if an error has occurred to then end the program.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Vulnerability | Instance ID | Location | Resolution |
| 1 | Buffer Overflows | 1-1 | File: Caesar.c  Function: fileWrite() | buffer = (char \*)malloc(file\_size + 1) |
| 1-2 | File: Caesar.c  Function: main() | dirArray = (char \*\*)malloc(length \* sizeof(char \*)) |
| 1-3 | File: Caesar.c  Function: main() | dirArray[i] = (char \*)malloc(MAXPATH \* sizeof(char)) |
| 2 | Integer Overflows | 2-1 | File: Caesar.c  Function: fileWrite() | if (file\_size < 0 || file\_size + 1 > SIZE\_MAX) |
| 3 | Command Injection | 3-1 | File: Caesar.c  Function: fileWrite() | file = fopen(argv[1], "rb") |
| 3-2 | File: Caesar.c  Function: fileWrite() | file = fopen(argv[1], "wb") |
| 3-3 | File: Caesar.c  Function: backupFile() | mkdir(backupDir, 0777) |
| 4 | Failure to Handle Errors Correctly | 4-1 | File: Caesar.c  Function: fileWrite() | if ((file = fopen(argv[1], "rb")) == NULL) |
| 4-2 | File: Caesar.c  Function: fileWrite() | if ((file = fopen(argv[1], "wb")) == NULL) |
| 4-3 | File: Caesar.c  Function: fileWrite() | if (buffer == NULL) |
| 4-4 | File: Caesar.c  Function: main() | if(directoryLength(dirName) == -1) return 1; |
| 5 | Catching Exceptions | 5-1 | File: Caesar.c  Function: fileWrite() | if ((file = fopen(argv[1], "rb")) == NULL) |
| 5-2 | File: Caesar.c  Function: fileWrite() | if ((file = fopen(argv[1], "wb")) == NULL) |
| 5-3 | File: Caesar.c  Function: fileWrite() | if (buffer == NULL) |
| 5-4 | File: Caesar.c  Function: fileWrite() | if (fread(buffer, 1, file\_size, file) != file\_size) |
| 5-5 | File: Caesar.c  Function: fileWrite() | if (scanf("%d", &shift) != 1 || (shift < Userinput\_INT\_MIN || shift > Userinput\_INT\_MAX)) |
| 5-6 | File: Caesar.c  Function: fileWrite() | if (fwrite(buffer, 1, file\_size, file) != file\_size) |
| 5-7 | File: Caesar.c  Function: isFile() | if (lstat(filename, &fileInfo) == -1) |
| 5-8 | File: Caesar.c  Function: directoryLength()  getFilenames() | if((dir\_ptr = opendir(dirName)) == 0) |
| 5-9 | File: Caesar.c  Function: backupFile() | if (originalFile == NULL) |
| 5-10 | File: Caesar.c  Function: backupFile() | if (backup\_file\_ptr == NULL) |