**Vulnerabilities**

Student Name

Course Name

Institution Name

Instructor’s Name

Date

**Vulnerability Analysis of file1.c**

**1. Buffer Overflow Vulnerability (processFile function)**

* **Type**: Buffer Overflow
* **How it works**:  
  The processFile function reads input from the file into a buffer using fscanf. The buffer is statically defined with a size of 64 bytes:

char buffer[MAX\_BUFFER];

The problem occurs because there is no check against the buffer having its bounds exceeded by input more than 64 bytes. It can corrupt nearby memory on the stack and result in unspecified behavior (Butt et al., 2022). It's a very typical buffer overflow example where the application does not perform length checks on an input value before using it.

In this case, fscanf It reads a string into the file, but if the string is more than 64 bytes long, it will fill the buffer and could harm important information in the program. If buffer overflow occurs, there is a risk that local variables or return addresses on the stack will be overwritten, so an attacker may gain control of the program.

* **GDB Exploit**:  
  To exploit this, an attacker could create a file data1.txt containing any word that takes more than 64 characters. Overloading the buffer will occur once the program has completed the file’s processing. For instance, if our input is as follows:

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

This input will overflow the buffer, possibly changing nearby memory and resulting in unexpected problems, for example, a crash or running unplanned code.

**2. Format String Vulnerability (extractResult function)**

* **Type**: Format String
* **How it works**:  
  The extractResult function in **file1.c** contains a format string vulnerability:

sprintf(items[count].word, "%d", num);

While this line may appear safe at first glance, the num variable is used directly in sprintf, which could allow an attacker to inject format specifiers (such as %x, %n, etc.) in the input data. This instance is referred to as a format string vulnerability. Providing a string that contains format specifiers allows an attacker to shape the program’s result or change the values stored in memory.

Format string vulnerabilities occur when user input is passed to functions like sprintf, printf, or fprintf they fail to check for proper format specifiers when filling in the pack. Hackers can use a malicious format string to access parts of memory they do not have access to (e.g., %x), or even writing to memory locations (e.g., %n).

* **GDB Exploit**:  
  A hacker can manipulate a file sent to the script so that it uses format specifiers. For example, by inserting %x%x%x%x, an attacker might be able to disclose the contents of the stack and direct how the code is executed.  
  For example, a malicious input file (data1.txt) might look like:

%x%x%x%x

The program would reveal details about its state when executed and printed the values on the stack.

**3. Heap Buffer Overflow (malloc in processFile)**

* **Type**: Heap Buffer Overflow
* **How it works**:  
  In **file1.c**, memory is dynamically allocated using malloc to store file contents. However, there’s no validation on the size of the file, meaning a very large file could lead to a heap buffer overflow:

file = malloc(filesize \* sizeof(char));

The application expects the file size to be normal, and if an unexpected malicious file is given to it with a file size that is greater, in the heap it can overflow the allocated memory. Heap overflow can result in a data-race environment and code execution if the attacker is in control of the heap layout.

This is a severe bug because heap overflows can be used to overwrite function pointers, return addresses, or other important structures, and thereby execute arbitrary code.

* **GDB Exploit**:  
  An attacker could take advantage of this flaw by using a very big file as input which can fill up all the memory reserved by the program (GeeksforGeeks, 2017). As a result, the heap may be corrupted, allowing malicious code to be executed by chance.
* For example, giving a file with a large number of lines could make the buffer overflow.

**4. Use After Free (freeItems function)**

* **Type**: Use After Free
* **How it works**:  
  In **file1.c**, the freeItems function frees the dynamically allocated memory for the word field in each DataItem. However, after freeing the memory, there’s no check to ensure that the pointer is not dereferenced again:

free(items[i].word);

After the memory is freed, if the program continues to use the pointer (e.g., accessing items[i].word), doing so will result in behavior that is not specified by the standard. A "use after free" issue is when the program accesses previously freed memory which can lead to memory corruption and security problems.

* **GDB Exploit**:  
  Attackers may use specific methods to cause the system to use variables that have already been freed. If an attacker influences the timing and manner of accessing the memory after it is freed, they might be able to take advantage of this flaw to run code of their own choosing.

With the extra memory in hand, an attacker could confuse the program, causing it to carry out unexpected tasks like reading or writing to wrong areas which could result in a crash or code execution.

**Vulnerability Analysis of file2.c**

**1. Buffer Overflow (display function)**

* **Type**: Buffer Overflow
* **How it works**:  
  The display function in **file2.c** copies the input string into the output buffer using strcpy:

strcpy(output, input);

However, there is no way to stop the process if the size of the input goes beyond the size of the output buffer (Klusáček, 2024). When the input string is bigger than the buffer size, the extra information will cause the buffer to overflow and could change other data nearby, possibly causing the program to malfunction.

* **GDB Exploit**:  
  The vulnerability allows attackers to insert input that is larger than the output buffer can store. For instance, an attacker could supply a value like:

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

This input would overflow the buffer and potentially corrupt adjacent memory, leading to program crashes or arbitrary code execution.

**2. Use of Uninitialized Variables (readFile function)**

* **Type**: Uninitialized Variable Usage
* **How it works**:  
  The readFile function uses the variable filesize, which keeps track of the file size, but it would be better if it was always initialized by the system. When a file is not read or opened in the correct way, filesize may have a corrupt or incorrect value which could lead to corrupted memory if it is used for any purpose (Pereira et al., 2021).
* **GDB Exploit**:  
  An unpredictable sequences of bytes in a malicious file can make the filesize variable invalid which may eventually result in inaccurate memory handling and a possible attack.

**3. Integer Overflow (readFile function)**

* **Type**: Integer Overflow
* **How it works**:  
  In **file2.c**, the filesize A variable can be set up so that memory is allocated whenever it’s needed. Nevertheless, if the files are exceptionally large, it might lead to a memory allocation error:

file = malloc(filesize \* sizeof(char));

If the filesize If the file is very large and causes a variable to overflow, it may result in the program failing to reserve sufficient memory which can lead to memory corruption or other weird actions.

* **GDB Exploit**:  
  An attacker could offer a file that results in the program failing to allocate memory properly due to an integer overflow. This might cause some memory to be corrupted or result in crashes.

**4. Improper File Handling (readFile function)**

* **Type**: File Handling Error
* **How it works**:  
  The program moves forward assuming the file is properly structured, even if it has not been validated closely. If the lines in the file are incorrect, it might result in memory corruption when the program reads them (Xu et al., 2022). Specific formats like ChunkID and ChunkSize are used by the program, yet it doesn’t check whether the data is in the correct format.
* **GDB Exploit**:  
  If an attacker gives a malicious file, the system can end up with unexpected actions such as memory problems or program failures due to handling it in the wrong way.

**5. Stack Buffer Overflow (filesize manipulation in readFile)**

* **Type**: Stack Buffer Overflow
* **How it works**:  
  The program uses a statically allocated buffer filename[24] to store the file name. An overflow caused by a filename that is longer than 24 characters may overwrite the values kept on the critical stack such as return addresses or other vital variables.
* **GDB Exploit**:  
  An attacker might use a filename with more than 24 characters to produce a stack buffer overflow. Such an action might make the stack unreliable and let the attacker control the program’s overall flow of commands.

References

Butt, M. A., Ajmal, Z., Khan, Z. I., Idrees, M., & Javed, Y. (2022). An In-Depth Survey of Bypassing Buffer Overflow Mitigation Techniques. *Applied Sciences*, *12*(13), 6702. https://doi.org/10.3390/app12136702

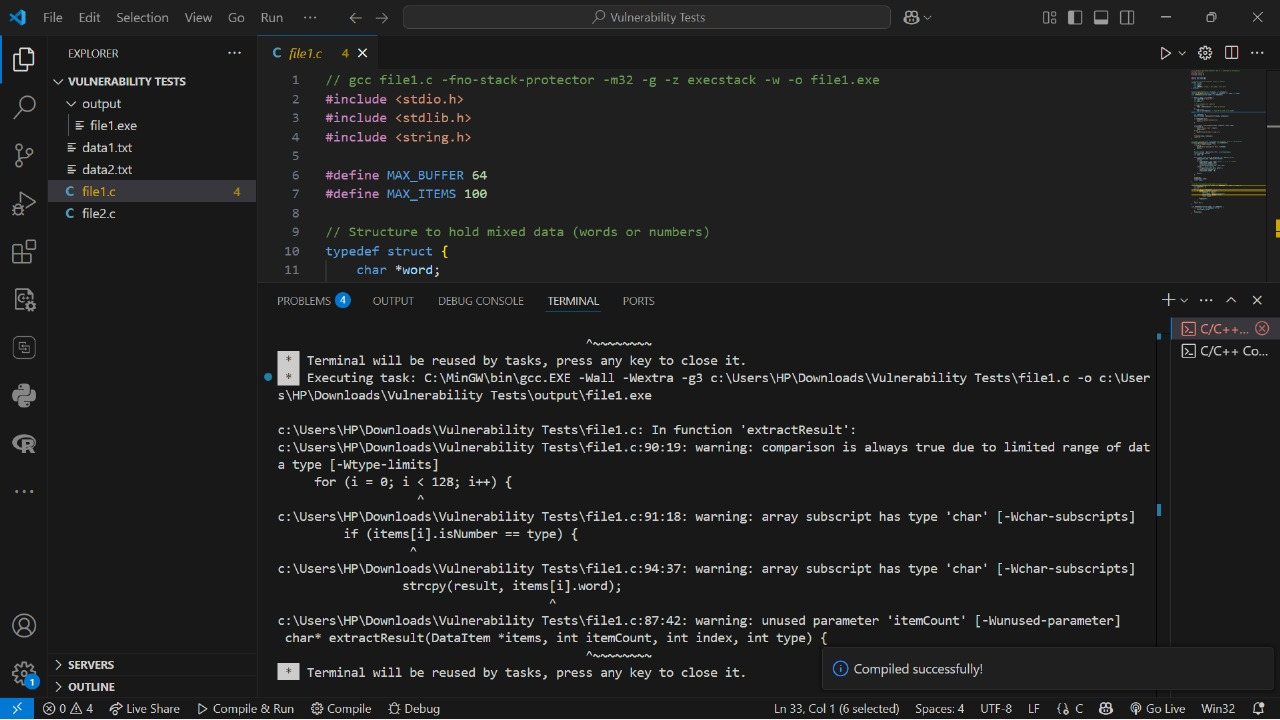
GeeksforGeeks. (2017, April 23). *Format String Vulnerability and Prevention with Example*. GeeksforGeeks. https://www.geeksforgeeks.org/format-string-vulnerability-and-prevention-with-example/

Klusáček, J. (2024). *Format String Vulnerability - Hacking Lab*. Hacking Lab. https://hackinglab.cz/en/blog/format-string-vulnerability/

Pereira, J. D., Ivaki, N., & Vieira, M. (2021). Characterizing Buffer Overflow Vulnerabilities in Large C/C++ Projects. *IEEE Access*, *9*, 142879–142892. https://doi.org/10.1109/ACCESS.2021.3120349

Xu, G., Lei, W., Gong, L., Liu, J., Bai, H., Chen, K., Wang, R., Wang, W., Liang, K., Wang, W., Meng, W., & Liu, S. (2022). UAF-GUARD: Defending the use-after-free exploits via fine-grained memory permission management. *Computers & Security*, *125*, 103048–103048. <https://doi.org/10.1016/j.cose.2022.103048>

File 1



File 2

