Shellcoding 101

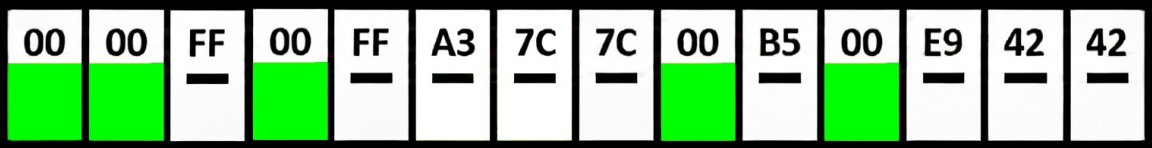


1. Introduction _____ 2
2. Bad Characters And Size _ 3
3. Debugging Your Shitcode _ 6

Introduction

A lot of the shellcodes available in public are actually quite good. However, the catch is that they aren't universal. Sometimes, we need to tweak them to fit our specific needs. Of course, no one is saying you should only write your own shellcode (even though that's pretty cool), but it's important to know how to create them. If we apply Occam's razor, we quickly realize that many of the issues we encounter when loading our shellcode stem from the shellcode itself. So, we need to ensure that our shellcode works perfectly. That's why it's essential to learn how to write them. In this paper, all the examples will focus on shell spawning shellcode. But really, you can apply these practices to any type of shellcode. We'll explore how to create them without bad characters, and we'll aim to make them more compact. Plus, we'll learn how to troubleshoot them effectively.

Bad Characters And Size



Let's take a look on this code:

```
section .text
global _start

_start:

mov  rbx, "/bin/sh"
push rbx
mov  rdi, rsp
push rax
mov  rsi, rsp
mov  rdx, 0x0
mov  rax, 59
syscall
```

It spawn a shell. But, Houston, we have a problem. If we take a look at our bytecode using *objdump -d shellcode*, we'll notice a bunch of null bytes.

Let's break down the first instruction, which involves a single null byte. The string "/bin/sh" has 7 characters, while RBX can hold up to 8 bytes. We don't completely fill RBX; we leave 1 byte empty, which results in that null byte appearing in our shellcode. We could try using "//bin/sh," but that won't work

because execve expects a null-terminated string. So, we need to load 8 bytes into RBX and then remove one to ensure it's null-terminated.

```
section .text
global _start

_start:

mov  rbx, "//bin/sh"
shr  rbx, 0x8
push rbx
mov  rdi, rsp
push rax
mov  rsi, rsp
mov  rdx, 0x0
mov  rax, 59
syscall
```

This example will work just fine because we're removing the last '/' symbol, so the file path remains correct, and the string is null-terminated. Now, let's dive into the other null bytes that come from the last two mov instructions. As we can see, we're moving 1 byte into our 8-byte registers. In the first example, we want to leave it empty. Instead of moving a null byte, you can simply use *xor rdx, rdx* .

Since most of the issues arise from moving values that are smaller than 8 bytes, we can take advantage of partial registers to access the lowest bytes of

the registers. For the last instruction, we can simply use: *mov al, 59* .

So our final result will be:

```
section .text
global _start

_start:

mov  rbx, "//bin/sh"
shr  rbx, 0x8
push rbx
mov  rdi, rsp
push rax
mov  rsi, rsp
xor  rdx, rdx
mov  al, 59
syscall
```

Alright, here we are! The shellcode is working perfectly, and guess what? No null bytes in sight!

Debugging Your Shitcode
Failed to load image

Keep in mind 2 main problems:

1. Logical Issues
2. RSP Adjustment

For the first issue, you really need to engage your brain, I'M SORRY ! As for the second one, it's important to remember that before making syscalls, returns, and calls, we have to keep our RSP properly aligned. This means that rsp % 16 == 0.

This problem can be a bit tricky. Depending on the contest, we might either fix or break our shellcode. Before you start fixing things, make sure you don't have anything important in your stack. If you do, save it in a register for later.

We have 3 main ways to adjust our RSP before any calls:

1. Jump to ret instruction
2. Allocate space
3. Adjust it by force

The first method is definitely the safer option. At the end of any function we have ret instruction, so we can replace return address to it, then put our shellcode address.

The second way is more risky but still works in some cases. You can check your stack size before making the call, adjust it using *sub rsp, value* , and then proceed. Just make sure to save the current RSP before you allocate or stored data in some other way.

The third method is a way more risky. It forces our RSP to be adjusted for the call, which could lead to issues down the line, so be cautious and avoid using the stack afterward.

IF YOU TOTALLY SURE: use *and rsp, -16* before call.

As a debugging tools use gdb with peda and strace. That will be totally enough.