****
* *
* Advanced Shellcoding Techniques - by Darawk *
* *
****

Introduction

This paper assumes a working knowledge of basic shellcoding techniques, and x86 assembly, I will not rehash these in this paper. I hope to teach you some of the lesser known shellcoding techniques that I have picked up, which will allow you to write smaller and better shellcodes. I do not claim to have invented any of these techniques, except for the one that uses the div instruction.

The multiplicity of mul

This technique was originally developed by Sorbo of darkircop.net. The mul instruction may, on the surface, seem mundane, and it's purpose obvious. However, when faced with the difficult challenge of shrinking your shellcode, it proves to be quite useful. First some background information on the mul instruction itself.

mul performs an unsigned multiply of two integers. It takes only one operand, the other is implicitly specified by the %eax register. So, a common mul instruction might look something like this:

```
movl $0x0a,%eax
mul $0x0a
```

This would multiply the value stored in %eax by the operand of mul, which in this case would be 10*10. The result is then implicitly stored in EDX:EAX. The result is stored over a span of two registers because it has the potential to be considerably larger than the previous value, possibly exceeding the capacity of a single register(this is also how floating points are stored in some cases, as an interesting sidenote).

So, now comes the ever-important question. How can we use these attributes to our advantage when writing shellcode? Well, let's think for a second, the instruction takes only one operand, therefore, since it is a very common instruction, it will generate only two bytes in our final shellcode. It multiplies whatever is passed to it by the value stored in %eax, and stores the value in both %edx and %eax, completely overwriting the contents of both registers, regardless of whether it is necessary to do so, in order to store the result of the multiplication. Let's put on our mathematician hats for a second, and consider this, what is the only possible result of a multiplication by 0? The answer, as you may have guessed, is 0. I think it's about time for some example code, so here it is:

```
xorl %ecx,%ecx
mul %ecx
```

What is this shellcode doing? Well, it 0's out the %ecx register using the xor instruction, so we now know that %ecx is 0. Then it does a mul %ecx, which as we just learned, multiplies it's operand by the value in %eax, and then proceeds to store the result of this multiplication in EDX:EAX. So, regardless of %eax's previous contents, %eax must now be 0. However that's not all, %edx is 0'd now too, because, even though no overflow occurs, it still overwrites the %edx register with the sign bit(left-most bit) of %eax. Using this

technique we can zero out three registers in only three bytes, whereas by any other method(that I know of) it would have taken at least six.

The div instruction

Div is very similar to mul, in that it takes only one operand and implicitly divides the operand by the value in %eax. Also like, mul it stores the result of the divide in %eax. Again, we will require the mathematical side of our brains to figure out how we can take advantage of this instruction. But first, let's think about what is normally stored in the %eax register. The %eax register holds the return value of functions and/or syscalls. Most syscalls that are used in shellcoding will return -1(on failure) or a positive value of some kind, only rarely will they return 0(though it does occur). So, if we know that after a syscall is performed, %eax will have a non-zero value, and that the instruction divl %eax will divide %eax by itself, and then store the result in %eax, we can say that executing the divl %eax instruction after a syscall will put the value 1 into %eax. So...how is this applicable to shellcoding? Well, their is another important thing that %eax is used for, and that is to pass the specific syscall that you would like to call to int $0x80. It just so happens that the syscall that corresponds to the value 1 is exit(). Now for an example:

```
xorl %ebx,%ebx
mul %ebx
push %edx
pushl   $0x3268732f
pushl   $0x6e69622f
mov %esp, %ebx
push %edx
push %ebx
mov %esp,%ecx
movb $0xb, %al  #execve() syscall, doesn't return at all unless it fails, in which case it returns -1
int $0x80

divl %eax  # -1 / -1 = 1
int $0x80
```

Now, we have a 3 byte exit function, where as before it was 5 bytes. However, there is a catch, what if a syscall does return 0? Well in the odd situation in which that could happen, you could do many different things, like inc %eax, dec %eax, not %eax anything that will make %eax non-zero. Some people say that exit's are not important in shellcode, because your code gets executed regardless of whether or not it exits cleanly. They are right too, if you really need to save 3 bytes to fit your shellcode in somewhere, the exit() isn't worth keeping. However, when your code does finish, it will try to execute whatever was after your last instruction, which will most likely produce a SIG ILL(illegal instruction) which is a rather odd error, and will be logged by the system. So, an exit() simply adds an extra layer of stealth to your exploit, so that even if it fails or you can't wipe all the logs, at least this part of your presence will be clear.

Unlocking the power of leal

The leal instruction is an often neglected instruction in shellcode, even though it is quite useful. Consider this short piece of shellcode.

```
xorl %ecx,%ecx
leal 0x10(%ecx),%eax
```

This will load the value 17 into eax, and clear all of the extraneous bits of eax. This occurs because the leal instruction loads a variable of the type long into it's desitination operand. In it's normal usage, this would load the address of a variable into a register, thus creating a pointer of sorts. However, since ecx is 0'd and 0+17=17, we load the value 17 into eax instead of any kind of actual address. In a normal shellcode we would do something like this, to accomplish the same thing:

```
xorl %eax,%eax
movb $0x10,%eax
```

I can hear you saying, but that shellcode is a byte shorter than the leal one, and you're quite right. However, in a real shellcode you may already have to 0 out a register like ecx(or any other register), so the xorl instruction in the leal shellcode isn't counted. Here's an example:

```
xorl    %eax,%eax
xorl    %ebx,%ebx
movb    $0x17,%al
int     $0x80

xorl %ebx,%ebx
leal 0x17(%ebx),%al
int $0x80
```

Both of these shellcodes call setuid(0), but one does it in 7 bytes while the other does it in 8. Again, I hear you saying but that's only one byte it doesn't make that much of a difference, and you're right, here it doesn't make much of a difference(except for in shellcode-size pissing contests =p), but when applied to much larger shellcodes, which have many function calls and need to do things like this frequently, it can save quite a bit of space.

Conclusion

I hope you all learned something, and will go out and apply your knowledge to create smaller and better shellcodes. If you know who invented the leal technique, please tell me and I will credit him/her.