Advanced Polymorphism Primer

VX Heaven

Dark Angel
40hex [11]
June 1993

With the recent proliferation of virus encryption "engines," I was inspired to write my own. In a few short weeks, I was able to construct one such routine which can hold its own. A polymorphic encryption routine is nothing more than a complex code generator. Writing such a routine, while not incredibly difficult, requires careful planning and perhaps more than a few false starts.

The utility of true polymorphism is, by now, an accepted fact. Scanning for the majority of viruses is a trivial task, involving merely the identification of a specific pattern of bytes in executable files. This approach is quick and may be used to detect nearly all known viruses. However, polymorphism throws a monkey wrench into the works. polymorphic viruses encode each copy of the virus with a different decryption routine. Since (theoretically) no bytes remain constant in each generated decryption routine, virus detectors cannot rely on a simple pattern match to locate these viruses. Instead, they are forced to use an algorithmic appproach susceptible to "false positives," misleading reports of the existence of the virus where it is not truly present. Creating a reliable algorithm to detect the polymorphic routine takes far more effort than isolating a usable scan string. Additionally, if a virus detector fails to find even one instance of the virus, then that single instance will remain undetected and spawn many more generations of the virus. Survival, of course, is the ultimate goal of the virus.

Before attempting to write a polymorphic routine, it is necessary to obtain a manual detailing the 80x86 instruction set. Without bit-level manipulation of the opcodes, any polymorphic routine will be of limited scope. The nice rigid structure of the 80x86 instruction set will be readily apparent after a simple perusal of the opcodes. Exploitation of this structured instruction set allows for the compact code generation routines which lie at the heart of every significant polymorphic routine.

After examining the structure of the opcodes, the basic organisation of the polymorphic routine should be laid out. Here, an understanding of the basics behind such routines is required. The traditional approach treats the decryption routine as a simple executable string, such as "BB1301B900022E8137123483C302E2F6." A true (advanced) polymorphic routine, by contrast, views the decryption routine as a conceptual algorithm, such as, "Set up a 'pointer' register, that is, the register whose contents hold a pointer to the memory to be decrypted. Set up a counter register. Use the pointer register to decrypt one byte. Update the pointer register. Decrement the count register, looping if it is not zero." Two routines which fit this algorithm follow:

Sample Encryption 1

```
        mov bx,offset startencrypt      ; here, bx is the 'pointer' register
        mov cx,viruslength / 2          ; and cx holds the # of iterations
decrypt_loop:
        xor word ptr [bx],12h           ; decrypt one word at a time
        inc bx                      ; update the pointer register to
        inc bx                      ; point to the next word
        loop decrypt_loop               ; and continue the decryption
startencrypt:
```

Sample Encryption 2

```
start:
        mov bx,viruslength              ; now bx holds the decryption length
        mov bp,offset start             ; bp is the 'pointer' register
decrypt_loop:
        add byte ptr [bp+0Ch],33h       ; bp+0Ch -> memory location to be
                                ; decrypted at each iteration
        inc bp                      ; update the pointer register
        dec bx                      ; and the count register
        jnz decrypt_loop                ; loop if still more to decrypt
```

The number of possibilities is essentially infinite. Naturally, treating the decryption as an algorithm rather than as an executable string greatly increases the flexibility in creating the actual routine. Various portions of the decryption algorithm may be tinkered with, allowing for further variations. Using the example above, one possible variation is to swap the order of the setup of the registers, i.e.

```
        mov cx,viruslength
        mov bx,offset startencrypt
```

in lieu of

```
mov bx,offset startencrypt
mov cx,viruslength
```

It is up to the individual to decide upon the specific variations which should be included in the polymorphic routine. Depending upon the nature of the variations and the structure of the polymorphic routine, each increase in power may be accompanied with only a minimal sacrifice in code length. The goal is for the routine to be capable of generating the greatest number of variations in the least amount of code. It is therefore desirable to write the polymorphic routine in a manner such that additional variations may be easily accommodated. Modularity is helpful in this respect, as the modest overhead is rapidly offset by substantial space savings.

The first step most polymorphic routines undergo is the determination of the precise variation which is to be encoded. For example, a polymorphic routine may decide that the decryption routine is to use word-length xor encryption with bx as the pointer register, dx as a container for the encryption value, and cx as the counter register. Once this information is known, the routine should be able to calculate the initial value of each variable. For example, if cx is the counter register for a byte-length encryption, then it should hold the virus length. To increase variability, the length of the encryption can be increased by a small, random amount. Note that some variables, in particular the pointer register, may not be known before encoding the rest of the routine. This detail is discussed below.

Of course, selecting the variables and registers will not in and of itself yield a valid decryption routine; the polymorphic routine must also encode the actual instructions to perform the job! The cheesiest polymorphic routines encode a single "mov" instruction for the assignment of a value to a register. The more complex routines encode a series of instructions which are functionally equivalent to the simple three byte "mov" statement yet far different in form. For example,

```
mov ax, 808h
```

could be replaced with

```
mov ax, 303h          ; ax = 303h
mov bx, 101h          ; bx = 101h
add ax, bx            ; ax = 404h
shl ax, 1             ; ax = 808h
```

Recall that the registers should be encoded in a random order. The counter variable, for example, should not always be the first to be encoded. Predictability, the bane of polymorphic routines, must be avoided at all costs.

After the registers are encoded, the actual decryption loop should then be encoded. The loop can perform a number of actions, the most significant of which should be to manipulate the memory location, i.e. the actual decryption instruction, and to update the pointer register, if necessary. Finally, the loop instruction itself should be encoded. This can take many forms, including "loop," "loopnz," "jnz," etc. Possible variations include altering the decryption value register and the counter register during each iteration.

This is the general pattern of encoding. By placing garbling, or "do-nothing," instructions between the essential pieces of code, further variability may be ensured. These instructions may take many forms. If the encoding routines are well-designed, the garbler can take advantage of the pre-existing code to generate null instructions, such as assignments to unused registers.

Once the decryption routine has been written, it is necessary to encrypt the virus code. The traditional approach gives the polymorphic routine the job of encrypting the code. The polymorphic routine should therefore "remember" how the precise variation used by the decryptor and adjust the encryption routine in a complementary fashion. An alternate approach is for the polymorphic routine to simultaneously encode both the encryption and decryption routines. Although it adds overhead to the code, it is an extremely flexible approach that easily accommodates variations which may be later introduced into the polymorphic routine.

Variable-length decryptors come at a significant trade-off; the exact start of the decryption cannot be known before encoding the decryptor. There are two approaches to working around this limitation. The first is to encode the pointer register in a single instruction, i.e. mov bx,185h and to patch the initial value once it is known. This is simplistic, though undesirable, as it decreases the variability of the routine. An alternate approach is to encode the encryption instruction in the form xor word ptr [bx+185h], cx (as in Sample encryption 2, above) instead of xor word ptr [bx], cx (as in Sample encryption 1). This increases the flexibility of the routine, as the initial value of the pointer register need not be any fixed value; correct decryption may be assured by adjusting the offset in the decryption instruction. It is then possible to encode the pointer register with multiple instructions, increasing flexibility. However, using either method alone increases the predictability of the generated code. A better approach would be to incorporate both methods into a single polymorphic routine and randomly selecting one during each run.

As an example of a polymorphic routine, I present DAME, Dark Angel's Multiple Encryptor and a simple virus which utilises it. They appear in the following article. DAME uses a variety of powerful techniques to achieve full polymorphism. Additionally, it is easy to enhance; both

the encoding routines and the garblers can be extended algorithmically with minimal effort. In the next issue, I will thoroughly comment and explain the various parts of DAME.