An Introduction to Encryption, Part III (Is an impenetrable encryption possible?)

vxug.fakedoma.in/archive/VxHeaven/lib/vmn06.html

MidNyte
Coderz [1]
February 2000

A short (and over-simplified) history of the virus

First of all came the un-encrypted virus. Then came virus scanners, which were basically just hex searchers looking for strings of hex only found in certain viruses. Viruses retaliated by coming up with encryption. Most of the virus is encrypted, and a small decryption engine at the start of the virus decrypts the virus body. As the encryption changes each time, the virus scanner is limited to searching for a much smaller section of code inside the constant decryptor. This wasn't much of a problem for virus scanners though. Viruses fought back again with polymorphism, this is essentially a way that a virus can change it's decryptor every time it infects a new file. That way no constant strings appear in the virus. Virus scanners came up with two ways to combat this, heuristics and emulation. Heuristics is simply looking for code that looks 'virus-like' This can be something as simple as the string '*.exe'. Emulation is the controlled running of the program instruction by instruction (not quite, but close enough for this article). A virus, under emulation, will be allowed to run just enough to decrypt itself and reveal it's code for either a straightforward scan or a generic (heuristic) scan. Anti-emulation is the viruses way of defeating this, it is a basically a way to detect emulation in progress and act accordingly. Some anti-emulation systems are incorporated into the decryptor of a virus, so that if the virus is being emulated it will not decrypt properly and hence not reveal it's code. Another defence the virus can use is anti-debugging, which is designed to hinder people who try to debug (in this case unencrypt) your code. This is different in that it doesn't defend the virus from antivirus programs, it defends it from the antivirus companies, the people who will try and study the virus and work out a way to detect it. Anti-debugging can be very simple, like turning off the keyboard interrupts at the start of the code and back on again at the end or it can be quite complicated, with the actual anti-debugging routine also being used as a key to decryption to protect against patching. This is the focus of this article.

Anti-debugging: more detail

Anti-debugging tricks are basically little pieces of code that have no overall effect on the running of a virus when being run as normal, but that cause the virus to malfunction, crash or worse when they are run under a debugging environment. The simple example above was to turn of the keyboard interrupt at the start of the code, and turn it on again at the end of the virus before control is passed back to the host program. This is simply achieved with:

```
in  al, 020h ; \
or  al, 002h ;  }Disable Keyboard interrupt
out 020h, al ; /

...at the start, and:

in  al, 020h ; \
and al, 0FDh ;  }Enable keyboard interrupt (FDh = NOT 2)
out 020h, al ; /
```

...at the end. When the virus is run under normal conditions, the keyboard is only off for a very small time, too small for people to notice. If the program is running under a debugger, as soon as the first few instructions are run the keyboard will no longer work, leaving the person at the debugger with no choice but to reset (at least it used to be in the good old days :) The simple work around for the person debugging was too simply patch over the code that turned off the keyboard with NOPs or other do-nothing instructions. Now the virus would work as normal under a debugger, without disabling the keyboard. To retaliate from this, the virus started to use it's anti-debugging routine as a key for decryption. The hex string to turn off the keyboard is 'E4 20 0C 02 E6 20'. If this was one of the decryption keys, the person debugging could not just replace the instructions with NOPs as this would change the key to '90 90 90 90 90 90' and cause the virus to decrypt incorrectly. This seems like an ideal solution, but unfortunately it is not. The whole point of this article is to point out the following fact: Any decryption routine can have it's basic functionality copied by someone determined to debug it. This means that your routine that uses an antidebugging routine and also uses that routine as a key for further decryption could be useless. Let's go through it with an example. The original virus looks like this:

```
start:
   in  al, 020h ; \
   or  al, 002h ;  }Disable Keyboard interrupt
   out 020h, al ; /

   xor si,si

   lea bx, start_of_encrypted
   lea cx, end_of_encrypted
   sub cx, bx
   shr cx, 001h

decrypt:
   mov ax, word ptr [start+si]
   xor [bx],ax
   inc si
   cmp si, offset decrypt
   jne next_key_word
   xor si,si

next_key_word:
   loop decrypt
```

The pointer to the relevant word of the decryption key is kept in si, and means that the key is all the code from 'start:' to 'decrypt:'. This works out as 'E4 20 0C 02 E6 20 33 F6 BB 19 01 B9 36 01 2B CB D1 E9'. If the keyboard part was nopped out the key would change to '90 90 90 90 90 90 33 F6 BB 19 01 B9 36 01 2B CB D1 E9', as we've already seen. What the person doing the debugging could do though, is simply take the encrypted portion of the virus and put it into his own program, only this time the key would be stored as data, not as an executable part of the program, like this:

```
start:
   xor si,si

   lea bx, start_of_encrypted
   lea cx, end_of_encrypted
   sub cx, bx
   shr cx, 001h

decrypt:
   mov ax, word ptr [key+si]
   xor [bx],ax
   inc si
   cmp si, offset key_end
   jne next_key_word
   xor si,si

next_key_word:
   loop decrypt

key:
   db 'E4 20 0C 02 E6 20 33 F6 BB 19 01 B9 36 01 2B CB D1 E9'

key_end:
```

As you can see, the above will decrypt the encrypted section in exactly the same manner, only because the key is stored as data we can change the code as much as we like.

Is an impenetrable encryption possible?

So then, is it possible to include enough current techniques, or to come up with a new technique to completely eliminate the chance of the antivirus programmers being able to decode it? Many people think that they have found a way to ensure that their program is completely impenetrable to decryption unless it is running at the time. This is, unfortunately, unachievable in theory. Because of the above demonstrated technique, any anti-debugging technique can be overcome by someone with enough time to debug a program by hand. This means that *any* anti-debug code you put into a virus can be got around eventually because the person debugging can always read what is going on in a hex editor and make a new routine to simulate it, hence the routine you write will not always be used to decrypt the code. They will only see one layer of decryption at a time, however, and this is the key to making in impenetrable encryption.

Conclusion

In the end then, we can never make it *impossible* for a researcher to decrypt a virus through programming tricks, however we can make it *impractical* through the use of scale, ie, we can use so many layers and different tricks that it is impractical to debug. If it takes a week for a programmer to decrypt a virus with hundreds of layers of encryption, they may be able to justify it. If they have ten viruses of this kind it gets harder to justify, and with a hundred of them it starts to get impractical. The ball would be back in their court.

```
                                        - MidNyte
```

As always, I welcome ANY feedback, good or bad, as long as it is reasonable.