VIRUS ANALYSIS 2

The Invirsible Man

Péter Ször
Symantec, USA

Over 25% of all *Windows* viruses were created during the first quarter of 2000. The number of 32-bit *Windows* viruses is now more than 450. Not surprisingly, several of the new ones show anti-heuristic characteristics.

The first generation *Windows* virus heuristics were extremely effective against viruses that target the Portable Executable (PE) format. It seems even virus writers were surprised by the results of relatively simply logic built into anti-virus products. Now that the initial phase of *Windows* virus development is over, and more complicated techniques are becoming evident to virus writers, more and more viruses are created which are difficult to detect (and/or repair) even with virus-specific detection methods.

Polymorphism was introduced into 32-bit *Windows* viruses very early. However, some of the polymorphic viruses are as easy to detect with today's technology as a regular virus. So, virus writers try to implement anti-emulation techniques since they are aware of the strongest component of modern anti-virus products: the emulator. This was true of DOS binary viruses and the same trend continues into 32-bit territory.

Anti-emulation techniques are often combined with slow polymorphism and entry point obscuring (inserting) methods. W95/SK and W32/CTX variants already show that detection and repair will be a more difficult issue this year. Most of these complicated viruses limit their lifetime, precisely because of their complexity. For instance, *SARC* (*Symantec Anti-virus Research Centre*) has only received one W95/SK submission so far. (In March 2000 alone, we received over 2000 submissions of the W32/PrettyPark worm!) It is fortunate that most virus writers do not seem to have noticed that complexity often kills a particular virus, and continue to create many viruses that have very little chance of survival in the wild.

At the beginning of March 2000, the latest edition of 29A's magazine was released to the public. This virus collection contains a large number of known 32-bit *Windows* viruses in source format, including the source of the W32/Ska worm. There are many unknown viruses in there too. One of them is W95/Invir.7051 – a real zoo virus, which uses many unique features that make it interesting to many anti-virus researchers.

At first glance the virus looks straightforwardly intentional, but it turns out that this is mostly related to its anti-heuristic feature. Moreover, a bug in the code limits the virus' replication to directories that start with \INF. Since the viral source was released in 29A magazine it is pretty clear that Invir's author had a plan to change \INF to \WIN, but forgot about it. Therefore, W95/Invir does not have the potential to cause any significant problems for users. However, I would like to examine the virus' anti-emulation trick by way of an introduction to these new methods that will make detection of future *Windows* viruses even more difficult.

Getting Control

Invir does not infect files by changing the entry point to point to the last PE section. That would make it very suspicious to a heuristic detector. The virus only infects PE files that have certain characteristics. Most importantly, the code section of the application needs to have a large enough slack area at its section end. (These slack areas are 'recycled' by many viruses, for instance by CIH variants).

Invir places a short polymorphic routine in this space which will eventually execute a polymorphic decryptor. The polymorphic decryptor is placed in the last section of the PE file together with the encrypted virus body – about 7-7.5 KB, depending of the size of the decryptor. The actual entry point will be modified to point to the first polymorphic routine in the code section of the PE host.

The first chunk of polymorphic code will calculate the entry point of the virus decryptor in the last section. However, this is dependent on a random condition. The virus either transfers control to the host program (original entry point) or gives control to the virus decryptor.

In other words, executing the virus does not guarantee that it gets loaded. Invir uses the FS:[0Ch] value as the random seed. On Win32 systems on *Intel* machines, the data block at FS:0 is known as the Thread Information Block (TIB). For instance, the DWORD value FS:[0] is a pointer to the exception handler chain. The WORD value FS:[0Ch] is called the W16TDB and is only valid under *Windows 9x*. *Windows NT* sets this value as 0.

When the value is 0, the virus will execute the host program. This is elegant – the virus will not try to load itself under *Windows NT*. Invir uses VxD functions to hook the file system and is therefore incompatible with *Windows NT/2000*. Executing the virus-infected executable will not cause an error message to be displayed under *Windows NT* and the host will be executed properly.

The W16TDB (FS:[0Ch]) is effectively random under *Windows 95*. The TIB is directly accessible without using an API. That is one of the simplest ways to get a random number. No additional (and more importantly, hard to mutate) code is necessary. (Using port commands would be an option, but again that would be incompatible with *Windows NT/2000*.)

The basic scheme of the first polymorphic block is the following:

```
MOV reg, FS:[0C]
AND reg, 8
ADD reg, jumptable
JMP [reg]
```

Garbage instructions are inserted into this, and some of the essential instructions are mutated to various forms. Any register can be used to hold the 'reg' value and make the calculation. A pointer is calculated and via that a redirection is made.

The problem is obvious for emulators. Without the proper value at FS:[0Ch], the virus decryptor will not be reached at all. It is a matter of complexity, and the detection of such viruses could be extremely difficult. Obviously, the virus writer wanted to create a difficult-to-detect virus and I am positive that some anti-virus products will not be able to detect W95/Invir for at least the foreseeable future.

The polymorphic decryptor uses multiple methods to encrypt the virus body with 32-bit keys. The virus is 'slow polymorphic' since it generates new keys only during installation in memory. The virus body is placed in the last section after the original data and the size of the last section is enlarged.

Going TSR and Infecting PE files

W95/Invir uses the CIH method to jump from User mode to Kernel mode without too much trouble. Just like W95/CIH, Invir also hooks the INT 3 (break-point) interrupt. In this way, the virus code becomes a little more difficult to trace in a debugger.

Invir gets the necessary API addresses first, then it checks if it is already active in memory. It compares the DWORD at the base address of KERNEL32.DLL plus 0x6c to .K3Y, and changes the text in the stub program to '*This program can not be run in Y3K.mode*.' Previously active copies patch the KERNEL32.DLL location with the virus ID.

The virus hooks the file system and monitors access to files. It tries to infect PE files during File Open, Attribute Check and Rename. It will not infect files in directories other than those that start with \INF – this is presumably because a code piece was not changed in the source before the virus was released in the 29A magazine.

Then the virus marks infected PE files with the dword value 0x79336B3F (y3k? in ascii) in the PE file header PointerToSymbolTable field to avoid multiple infections. The last section's characteristic field is modified to include the writeable attribute. Invir got its name from the text that can be found only after decryption:

```
You can not find what you can not see.
Invirsible by Bhunji (Shadow VX)
```

So, what are the possibilities of detecting such viruses?

Detecting Invir

Basically, the detection of W95/Invir can be almost as complicated as the detection of entry point-obscuring viruses. The first obvious solution is virus-specific detection on an anti-virus source level. Many anti-virus products use this method but they cannot be updated in a matter of just a few hours.

Moreover, additional porting issues will make the procedure even slower. If anti-virus researchers are not completely free to control the emulator (if there is any) of the product, they are in trouble. The emulator's environment needs to be freely controlled and this way a virus-specific emulator session can solve the decryption easily.

Cryptographic methods can also be used in order to decrypt the virus body. Such a method is already being used by various anti-virus products nowadays. Cryptographic detection needs proper examination of the polymorphic engine of the virus.

Since W95/Invir does not always compile (yes, the polymorphic engine has its own compiler!) a valid polymorphic decryptor, the virus sometimes fails to decrypt itself properly. Only those products that use cryptographic detection will be able to deal with this slight problem. (A similar problem existed back in the DOS polymorphic days with viruses such as the Hare family.)

Conclusion

As virus writers use more anti-emulation tricks to challenge anti-virus vendors, the problem of detecting a particular virus becomes more and more difficult. The author of W95/Invir has plans to use EPO techniques in his next release, as well as incorporating mass-mailing capabilities.

W95/Invir

Aliases: W95/Invirsible.

Type: *Windows 95* PE infector.

Interception: Hook on IFS.

Hex Pattern in Exe Files:
Not possible – the virus is polymorphic.

Self-recognition in Memory:
KERNEL32.DLL's base address + 0x6c is modified to hold the DWORD value of '3YK'.

Self-recognition in Files:
The PointertoSymbolTable field of the PE header is modified to hold the DWORD value of 'y3k?'.

Removal: Delete infected files and replace them from backups.