A Discussion Of Methodology And Implementation / Buz[FS]

Polymorphism
A discussion of metodology and implementation
by Buz [FS]
from *Zine #2

This article deals with a viral technology that has been widely documented, discussed and implemented. However, it is aimed at explaining certain design flaws in current polymorphic engines and proposing solutions for these flaws, as well as suggesting improvements to current technology.
The discussion will present an overview of the history of polymorphism pertinent to our subject, anti-virus detection methods, and will present concepts needed for properly designing polymorphic engines with a view to their survival in the wild. It will also include a section on structuring and writing polymorphic engines.

The Evolution Of Polymorphic Engines And Their Significance.

The history of polymorphism began with experimentation. Virus authors recognised the susceptibility of their viruses to scan strings and encrypted their code. Even then, the decryptors were fixed, so anti-virus software generally had little trouble with a virus that was analysed and for which a scan string was extracted. A number of authors would rewrite their viruses to create strains which weren't scanned for at the time. A select few, however, started experimenting with new technology. A German programmer going by the handle of ~knzyvo} implemented dynamic encryption into his Fish family. The Whale virus, however was a more notable event. 30 different encryptors were used for this virus, which meant the anti-virus researchers had to include multiple scan strings. Dark Avenger's Phoenix family would modify bytes of their own decryptors, thus forcing anti-virus software to use wildcard scan strings. An American anti-virus researcher named Mark Washburn wrote a family of viruses that would generate a different decryptor altogether for every time the virus would replicate.

The real breakthrough in polymorphism was, though, the release of Dark Avenger's Mutation Engine, or MtE. This engine was distributed in a form of an object linkable to a file, and was what started the revolution in the way viruses were written. Anti-virus researchers were at a loss. The traditional methods of detection were obsolete, since this engine would have needed 4.2 billion signatures, many of which might be present in legitimate programs. Instead, most anti-virus researchers opted for methods like algorithmic scanning - checking whether or not code in question could be produced by a polymorphic engine. Several months later, anti-virus software couldn't reliably detect MtE-generated decryptors.

A second blow came to the anti-virus industry with the release of Trident Polymorphic Engine, written by Masud Khafir. A more complex algorithm was used for producing encryptors, and again, anti-virus researchers were left with the task of reliably detecting TPE. While the decryptors themselves weren't particularly sophisticated, they could easily be mistaken for encryption used in commercial software, and later, several other engines would be mistaken for TPE samples.

A new concept was introduced in 1993. Neurobasher's new Tremor virus spread widely in Germany. It seemed to researchers that a suitable algorithm was devised for its detection, yet, the virus continued to elude scannes in the wild. After thorough analysis of the virus's code, it was found that instead of generating random numbers, Tremor would use relatively immutable data to create its permutations. New strains would be generated every, say, full moon or on infecting a new system. This meant that the anti-virus researchers would need to spend even more time and effort on analysing a polymorphic virus lest they release an incomplete algorithm.

Meanwhile, across the channel, a British virus writer known as the Black Baron released his polymorphic viruses built around an engine called SMEG. This engine introduced the concept of generating decryptors with large amounts of junk instructions present in the decryptors. Once again, scanners had difficulty when confronted by a new polymorphic beast. It took a much longer time to analyse a piece of code and determine whether or not it was encrypted by SMEG by picking out the decryptor from the junk.
[MGL's note: If you take closer look on SMEG, you will get the point - generated decryptors are huuuuge]

From 1992 to 1994, an unknown researcher in Uruguay busily created a family of 10 viruses, each more polymorphic than the last. The novelty of his approach rested in tracking the code that was generated, and producing decryptors that looked even more like the real thing. It became difficult to distinguish polymorphic decryptors from real code.

Another 1994 engine that made a significant impact on the anti-virus industry was DSCE. Dark Slayer stated that his decryptors contained no loop, key, or start-up values. In a way, he was correct. However, it's an exaggeration of what the engine really did - these structures were concealed in a massive (at the time) decryptor by point-encrypting the opcodes that resembled a decryptor loop. Once again, scanners were slowed down by having to analyse the decryptor in depth.

While there are several other polymorphic engines just as technically advanced as those mentioned above and the authors of which deserve just as much recognition, these are the ones that we need to illustrate the design of a solidly built polymorphic engine.

Polymorphic Virus Detection Methods.

So, what methods are used to detect polymorphic viruses in the wild? And what weaknesses of the polymorphic engine design do they exploit? These are questions particularly interesting to any aspiring writer of a polymorphic engine. It must be understood that anti-virus software developers often implement the lowest-grade working solutions. For instance, when Whale appeared, multiple scan strings were used instead of an algorithm. When MtE appeared, an algorithm was used instead of more sophisticated methods of analysis such as single-stepping through the decryptor or emulation of the decryptor code. So, a virus sufficiently advanced to defeat currently available methods of detection would instantly get a time window that would give it a chance to spread in the wild. Well, let's take a look at what we're up against.

Polymorphic Virus Detection Countermeasures.

A properly designed engine should aim to generate code that is as obscure and difficult to detect as possible. Here's a simple point-by-point guide to stopping most detection methods.

An example time-out attack could be orchestrated in the following fashion. The virus is encrypted and written to disk, but the key is not saved. To derive the key, some sort of checksum of the unencrypted code is saved. The virus is decrypted with a random key, the checksum is calculated, and the two checksums are compared. If the two checksums do not match, the virus is re-encrypted with the reverse operation and the process is looped back. This makes for a larger, more sophisticated loop, which an emulator must go through hundreds of times, magnifying the relative slowdown. Anti-virus emulators are built with avoiding infinite loops in mind, so perhaps an emulator will skip such a structure.
[MGL's note: For example Spanska's IDEA.6126 uses above described approach]

Another time out strategy is building complex decryptors. This will be further explained in the section dedicated to engine structure, but the premise is that the more code the emulator has to execute, the slower it will be. Therefore, a decryptor containing a moderate number of conditional jumps, calls to subroutines, and other such structures will be slower to emulate than one that's purely linear.

Designing And Structuring A Polymorphic Engine.

A polymorphic engine is no trivial task to write. Much of the overhead can be reduced by setting down an appropriate structure for the engine and organising it according to that.
The function of a polymorphic engine is to encrypt a piece of code and produce a decryptor that will then decrypt the encrypted code. The decryptor that is produced must be as variable as possible. To achieve this, and to make analysis more difficult, a polymorphic engine will usually be written to produce:

Here, I would like to both compliment a virus writer for his achievement and expand on his idea to suggest a new design standard for advanced polymorphic engines. Almost 4 years ago, a virus was published in an underground virus exchange e-zine called 40Hex. The name of

this virus was Level-3, and the author was then-famous Vyvojar, who had by then firmly established himself with the notable One_Half virus.
[MGL's note: according the Vyvojar One_Half virus was written to demonstrate virus with maximum spreading abilities while One_Half successor Level-3 was demonstrating use of hardcore poly encryption.]
The design of the engine was revolutionary - the engine would generate the decryptor code, and then emulate it to determine the instruction flow. This concept is quite similar to the ideas I was working on at thetime, which leads me into the design structure of an engine that would be extremely resistant to most analysis methods.

First of all, all of the code the engine generates would have to be emulated by its own internal emulator. This means the contents of the registers can be quite easily tracked by the emulator and the levels of complexity will be increased to a great degree. For instance, when a value like a key, start of the encrypted area, or any such area is required, the engine can quite simply fix up the values already held in the registers. The values on the stack would be emulated too. The possibilities here are really much bigger than the simple variation that can be achieved by setting down sets of rules for generating code.

Secondly, all of the 8086 opcodes should be produced by the engine. However, they should be produced in different frequencies - for instance, an average decryptor would usually contain about 80% of the 8086 instruction set, with the remaining 20% generated in 1 out of 20 samples. The garbage generation can be handled by building tables which would be accessed with different probabilities. Of course, producing 80386+ opcodes, or floating point coprocessor instructions would increase both variability and make the engine harder to emulate. Remember, no emulator is perfect, and most anti-virus emulators cannot handle complex instruction sets in decryptors.

Thridly, the structure of the decryptor itself should be complicated by such things as calls and conditional jumps. The reason for this is quite simple - it facilitates emulator slowdown. For example, 3 calls to a 20-byte subroutine are equivalent to 69 bytes of code. Conditional jumps are very useful for slowing down the process too. Emulators will attempt to emulate every path that is available if it cannot be predict the direction of the jump - a technique known as path emulation. One jump that cannot be predicted by an emulator means the decryptor will have to be emulated twice. Two such jumps mean the decryptor will have to be emulated four times. Structures like this ensure that a small decryptor may take as long to emulate as a very large decryptor.

Finally, a word about layers. It seems that a lot of people believe a higher number of layers will ensure adequate protection. This protection is only there in so far as the emulator will simply take as long to emulate the layers as it would for a single decryptor of the collective size of these layers. There is a restriction on the largest possible size or the largest number of layers that has to be made, and it seems optimal to maintain only two layers, one to fool heuristic scanners into thinking it's legitimate code and decrypt the second one, and the second being a simple cyclical decryptor for the rest of the virus.

I hope that this has given you an insight, insiration or ideas to implement. Good luck with designing your new super-polymorph. ;)

Special thanks to MGL, Pockets and Owl for their invaluable ideas and suggestions.

Greetings fly out to all my friends in the scene.