Polymorphic Viruses: Implementation, Detection and Protection

Polymorphic Viruses:
Implementation, Detection and Protection.
by Tarkan Yetiser

```
virus entry point:
Polymorphic Decryptor
*****************************
** encrypted ****************
** main virus body **********
*****************************
*****************************
```

This paper discusses the subject of polymorphic engines and viruses. It looks at general characteristics of polymorphism as currently implemented. It tries to maintain a practical presentation of the subject matter rather than an academic and abstract approach that would confuse many people. Basic knowledge of the Intel 80x86 instruction set will be highly useful in understanding the material presented. A very detailed discussion is avoided not to have the side effect of "teaching" how to create polymorphic engines or viruses. The purpose is to help computer professionals understand this trend of virus development and the threats it poses. It should serve as a starting point for individuals who would like to get an idea about the polymorphic viruses and how they are implemented. Long gone are the days of innocence, when any schoolboy could write a virus scanner using a few signatures extracted from captured virus samples.

The subject of polymorphism can be extended to other areas such as anti-reverse-engineering or anti-direct-attacks, and it can be argued to be useful in that context. This paper only looks at the use of polymorphism in PC viruses to avoid simple detection techniques.

1. Introduction

In the Spring of 1992, we analyzed a polymorphic engine called MtE, and provided a report to satisfy the curiosity of the public. We also provided a freeware program called CatchMtE. At the end of 1992, we received reports of a new polymorphic engine, called TridenT Polymorphic Engine, or TPE for short. It was released in Europe by someone who calls himself Masud Khafir.

Both MtE and TPE are distributed as object modules that can be linked to programs allowing them to create different-looking decryptors. One notorious use of such a module is for writing so-called "polymorphic" viruses. Prior to the appearance of TPE, several viruses using

MtE (Mutation Engine) have been seen. The claimed author of TPE pays tribute to the author of MtE in the documentation that comes with TPE.

MtE is about 2.4 kilobytes in size. It can generate decyptors using based or indexed addressing modes with word-size displacement. The decryptor steps through the code a word at a time. It uses 4 variations of the based or indexed addressing modes. The structure of the decryptors is constant.

TPE is about 1.5 kilobytes in size. It can generate decryptors using based or indexed addressing modes with or without displacement. Unlike MtE, TPE can also create byte-at-a-time decryptors, as well as word-at-a-time. It also uses more addressing modes available on the 80x86; 6 variations of the based or indexed addressing modes are used. Its more general nature makes TPE less predictable, and complicates the task of recognizing TPE-based viruses. Many encryptive viruses can be considered a subset of TPE-based decryptors, and may be flagged as such. To overcome this problem, one has to check for other viruses before performing check for the presence of a TPE decryptor.

2. Polymorphism and Its Common Use

We will now present some preliminary information on polymorphism in general, and discuss certain features of the Intel 80x86 instruction set.

Although polymorphism is independent of encryption, it is easier to use encryption to hide the main body of the virus and implement a polymorphic decryptor. Viruses aim to keep their size as small as possible and it is impractical to make the main virus body polymorphic. One could attempt to rearrange the instructions in the main virus body or even use different instructions (to defy recognition techniques based on checksumming). Such an effort would not be as helpful or as easy as the common approach of using encryption in combination with polymorphism. In summary, current polymorphic viruses keep the main virus body encrypted, and implement a polymorphic decryption routine in plaintext. Since the decryptor is comparatively small, and performs one specific task, the amount of time and effort needed to craft a polymorphic virus is significantly reduced. Pictorially, a generic polymorphic virus would be structured as follows:

3. Implementation of Polymorphism on the Intel 80x86

In almost every case we have examined, the polymorphic engine exploits the fact that certain computations can be performed using different registers and instructions. To step through encrypted portion of code, for example, one can use DI, SI, or BX registers. To increment or to decrement the index value, one can ADD to the index register, INC it, or use an implicit instruction that increments it (CMPSB is used in TPE for example).

Polymorphic engines can also rely on the availability of instructions that are coded using the same opcodes. On the 80x86, there are 11 opcodes used for several different instructions: 80, 81, 83, D0, D1, D2, D3, F6, F7, FE, and FF. When it is necessary to encode information about one operand, the middle three bits of the ModRM byte are used to distinguish operations. The ModRM byte follows some opcodes and can either extend the opcode or contain information about the operand and addressing modes of an instruction. It contains three fields: Mod (2 bits), Reg (3 bits), and R/M (3 bits). For example, for the opcode 80, the middle three bits of the ModRM byte, which make up the Reg field, have the following meanings:

The second operand of each instruction with an opcode of 80 is an immediate byte, so the ModRM fields in the second byte of machine code encode the first register or memory operand.

By flipping a few bits, it is possible to generate code achieving many different operations easily. Combined with a rich set of addressing modes, and a good random number generator, one can create very different-looking decryptors.

4. Common Characteristics of Polymorphic Viruses

Polymorphic viruses of varying degrees of complexity have appeared in the past. In the anti-virus community, polymorphism is still an active area of discussion and much disagreement. Most researchers would agree that certain viruses are simply encryptive with a variable key. We do not consider such beasts polymorphic since they can be recognized using simple wild card scan strings. The number of such viruses significantly increased over the past few years as a reaction to the worldwide availability of good signature-based virus scanners. Similarly, scanners also evolved to handle such beasts using more flexible signatures that can include wild card or don't-care values to accommodate the variable parts of the decryptors.

There are other viruses that exhibit some polymorphic behavior, though they remain unsophisticated. Classification of such beasts is a gray area. For example, some viruses can generate a limited number of different-looking decryptors. These do not implement a polymorphic code generation engine, but rather pick from a pre-computed set of code fragments that they carry along. Writing detection routines for such viruses is not too difficult.

To aid in implementing polymorphism, a random-number generator is often used. A random-number generator provides selection of a part of the decryptor. This selection could be for "noise" instructions that are inserted in the decryptor to render signature-based scanning ineffective. It could also be used to select a certain addressing mode and appropriate registers. The obvious use of random number generators is to get a seed value for encryption.

Another aspect of a polymorphic engine is the choice of instructions that actually perform the decryption. XOR is a favorite choice. The chosen operation modifies the code to be decrypted using an addressing mode that allows external memory access. By external, we mean outside the processor registers. By using several instructions and many different addressing modes, a polymorphic engine can achieve a large number of combinations of decryptors.

Usually, a loop construct is set up to step through the encrypted code. On the 80x86, many instructions can be used to accomplish looping. The loop counter can be modified implicitly using LOOP instruction. Or it could be more explicit using a DEC CX, JNZ ?? combination. Several such possibilities exist. Again, availability of different approaches increases the number of appearances a decryptor can have. In the MtE, the loop construct was very predictable not only because it used a specific instruction but also because it had a characteristic that made it very simple to recognize MtE-based decryptors. Although "appearance-based" analysis on the MtE left many anti-virus developers in despair, a structural analysis proved to be very effective. We doubt even the developer of MtE noticed how predictable his polymorphic engine is.

The goal of a polymorphic engine is not to leave any predictable sequence of instructions that a virus scanner can use to simplify or optimize an appropriate detection algorithm. For example, using the same instruction to increment the index register is too predictable. A mixture of instructions that achieve the same result explicitly or implicitly (as in TPE) would make detection a lot harder.

5. Detection of Polymorphic Viruses

There are several problems that must be addressed to develop a reliable detection routine for a given polymorphic engine. The most challenging one appears to be avoiding false positives. Many programs include a run-time decompressor to reduce space requirements. When loaded in memory, the decompressor takes charge and produces an expanded copy of the code that can be run on the CPU. Furthermore, some programs are actually encrypted on disk, and they are decrypted on the fly when they are loaded. The purpose of using such a scheme is to make reverse-engineering efforts difficult. Such programs are more likely to trigger a false positive.

To reduce false positive rate, one can limit the search to a small area of the suspect file. This would be helpful in many cases; however, the compressed and encrypted files carry their decompressor/decryptor at the entry point of the program, just where many viruses plant their code. Another trick a few viruses employ is to hide the virus entry point where the decryptor is located. Of course, scattering the virus code while keeping the host functional complicates the virus.

A structural analysis of TPE reveals some information that can be used to recognize a TPE-based decryptor, although it is not as useful as it is in the case of MtE. Note that structural analysis does not rely on presence of specific byte sequences, and therefore offers a powerful tool that can be used in developing recognition routine for many polymorphic engines.

6. Protection Mechanisms and Solutions

Anti-virus field is full of solutions to almost every existing virus problem, and sometimes solutions to non-existing problems as well. Among other things, one solution prevailed as the most popular: scanning. The inherent flaw in this solution is that it cannot cope with viruses that it does not have signatures for. Therefore ugrades are issued frequently. Deployment of such upgrades in large installations requires tremendous effort, and quickly translates into having a very old version of a scanner installed; even then not necessarily used. Aside from the false sense of security scanning gives, some companies make less-than-honest claims about the capabilities of their scanners to promote their products. Testing such claims is beyond the resources of most organizations. Those who could conduct such tests are not always impartial.

Scanning has its place, and it is very useful when used properly. The best protection against computer viruses is user awareness and education. Unfortunately, a well-written virus can be so transparent that even the most observant user may not notice a thing. Even worse, many users lack the technical knowledge to understand how their computers work. Most people simply do not have the time or desire to learn more than how to use a mouse. The proof of this is the proliferation of products that make excessive hardware demands without offering improved functionality. The bulk of the code takes care of the user interface. This trend is unlikely to change.

More powerful techniques such as integrity checking can deal with viruses of different kinds, including polymorphic viruses. Even a simple integrity checker should be able to find if a polymorphic virus is spreading all over your disk. In other words, solution to polymorphic viral spread has been available for quite some time. Note that there are some viruses that target integrity-based products and they must be dealt with accordingly. A detailed discussion of such viral methods can be found in a paper written by anti-virus researcher Mr. Vesselin Bontchev of Virus Test Center at the University of Hamburg. His paper is titled Attacks Against Integrity Checkers, and it is available via anonymous FTP. Interested individuals are encouraged to read his excellent paper.

The point is that an integrity-based anti-viral solution should be made part of your arsenal against viruses. Such a solution could easily provide you with early detection of viruses before they spread. Once the viral spread is controlled, viruses become nothing more than another "computer glitch" that can haunt only those without timely backups.

We believe that neither harsh legislation nor emphasis on responsible computing can stop virus development, although they may slow it down. It is necessary to take matters into your own hands and protect your computers adequately.