

Experiences with Implementing a Simple Antivirus Engine *

Boris Nechaev

boris.nechaev@hiit.fi

Helsinki Institute for Information Technology HIIT / Aalto University

June 7, 2010

Abstract

In this report we describe our experiences with implementing Simple AntiVirus Engine (SAVE) designed to statically detect a virus family in obfuscated files. The engine is implemented in a flexible and extensible way allowing for easy incorporation of new features. We pay much attention to assessing advantages and drawbacks of the current detection scheme along with proposing possible ways of improving the engine.

1 Introduction

Implementation of an efficient and robust antivirus engine is a challenging task. There are many requirements imposed on a modern antivirus engine - good performance, reliable detection, low rate of false positives and false negatives, maintenance of an up-to-date virus fingerprint database, possibility of proactive detection, etc. As in many other complex endeavors it may be hard to fulfill all these requirements to the full extent. Therefore it is necessary to carefully consider various trade-offs. For instance, performance of a solution is usually in conflict with its generality, i.e. one can either implement fast and efficient detection of a single virus based on its peculiarities or a general but inefficient detection engine capable of recognizing many families of viruses, but hardly both at the same time. Though in some cases it is possible to discern common features of a single virus family and use them to detect the family efficiently.

This report is structured as follows. In Section 2 we describe internals and behavior of the virus that our engine is designed to detect. Then, in Section 3 we describe architecture of the engine. Finally, in Section 4 we evaluate strengths and drawbacks of the current implementation and speculate about future work and possible enhancements of the engine.

2 Virus description

Before implementing a detector of a particular virus it is necessary to study its behavior, structure, changes it makes to the infected file, actions it performs in the operating system, etc. This knowledge will give clues on what fingerprints to use in order to reliably detect the virus.

We start with a brief description of the virus we call "Win32/T-110.6220". The first easily observable effect of the virus is that when the infected executable is run, it outputs the "You must detect this file" string instead of the "You must *NOT* detect this file" string output by benign programs.

The next step is to examine PE header of an infected executable and compare it to the PE header of a clean file. Inspection shows that all `badN.exe` files are very similar to each other, and so are all the `goodN.exe` files. However, the former have entry point address, section sizes and import table different from the latter. Though many aspects such as section names remain the same.

A more in-depth inspection requires disassembling and reverse engineering the files. This procedure shows that all the files are obfuscated with a simple obfuscator. First indication of this is that the output strings are not seen in plain text in the file assembly code. Second, the file

*T-110.6220 Malware Analysis and Antivirus Technologies course project, Spring 2010.

contains many consecutive bytes that are meaningless, i.e. the disassembler fails to convert them to a meaningful set of assembly code mnemonics. We further call this set of consecutive bytes an obfuscated body. Finally, running the program in debugger suggests that printing out of the strings is preceded by decoding the obfuscated body.

Decoding procedure consists of three parts. In the first, deobfuscation parameters — obfuscated body address, obfuscated body length and obfuscation key — are initialized. Then, byte-wise deobfuscation is performed in a loop. Each byte is XOR'ed with the previous obfuscated byte; the very first byte is XOR'ed with the key mentioned above. Finally, control goes to a point inside the deobfuscated body, which can be viewed as a real entry point of the program.

Behavior of deobfuscated body also differs between `badN.exe` and `goodN.exe` files. The former in addition to printing out the strings also creates a mutex.

3 Architecture

In the light of the constant arms race between virus writers and antivirus vendors it is extremely important to assure evolvability and extensibility of antivirus products that will enable easy incorporation of new features and advanced technologies. For this reason we designed our antivirus engine in a modular fashion so that separate modules can be added, removed or altered according to current needs.

Figure 1 depicts the architecture of the engine which consists of six major components - I/O abstraction layer, PE header parser, deobfuscator, analyzer, logger and fingerprint database. We further discuss each in detail. It must also be noted that in most cases it will be trivial to add other components such as archive decompressor or emulator into the workflow.

Current version of SAVE works with non-archived files saved on disk. However, the I/O abstraction layer makes it possible to accept data from other channels. For instance, it may be possible to implement pipeline, network socket or memory dump input. In essence, the I/O abstraction layer converts any form of input into a universal bytestream that can be further used by other components of the engine. However, we decided not to give other components direct access to the stream data structure since

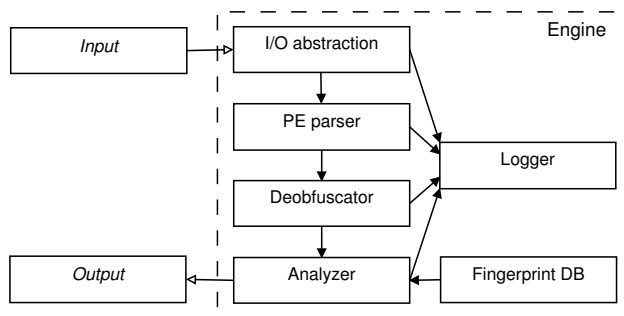


Figure 1: SAVE architecture.

this would make access operations too input type specific. Instead, the I/O abstraction class provides stream manipulation operations such as reading a single byte, number of bytes, the whole file or setting the pointer to an arbitrary position in the stream. It must also be noted that some of input types don't readily support all the above features. For instance, network socket doesn't allow seek operation and therefore it must be implemented in a workaround manner.

The first virus detection step is assessing the PE header of the file. If it is missing, the file is reported to be non-executable and the engine proceeds to the next file. The PE parser class also provides a set of features that can be further used by deobfuscator or analyzer. For instance, it has a function that extracts and returns the program entry point address. Another important operation is calculation of the difference between an address in memory and on disk. This is needed because in the file on disk some of the addresses (e.g. the address of obfuscated body) are stored as memory addresses. Thus in order to map the latter to an offset on disk, one must calculate the above difference and subtract it from the hard-coded memory address. Finally, we use the PE parser to return names of the functions imported by the executable.

Following the description of obfuscation scheme in Section 2 it is possible to implement a deobfuscation algorithm. The algorithm is designed in a generic way and therefore is able to decode all files encoded with the same scheme. First, starting at the entry point of the program it searches for three parameters needed for decoding - obfuscation body address, obfuscation body size and obfuscation key. If these parameters are not found after a num-

ber of instructions, the deobfuscator reports that the file is not obfuscated. If the parameters are extracted successfully, deobfuscation is done by XOR'ing the bytes in the obfuscated body with respective keys. The decoded body is further passed to the analyzer.

The analyzer module's task is to determine whether the file is infected. It does this by trying to match the fingerprints stored in the fingerprint database. The database consists of entries with three fields - virus name, fingerprint type and fingerprint value. The fingerprint type says how to match the fingerprint value. In the current implementation there are two types - 'Import' and 'String'. The first checks whether the fingerprint value is among the imported function names, while the second matches the given string versus the whole deobfuscated body. Currently we use one imported function name and two strings that are present in `badN.exe` files, but not in `goodN.exe` files - 'CreateMutexA', 'You must detect this file' and 'bad_mutex'. Thus our current detection scheme is based on three fingerprints represented as three rules in the database.

The remaining auxiliary component of SAVE is the logger. All other classes are using it to save intermediary and debug information about the file as the analysis proceeds. The logger automatically attaches current date and time to each log message, which may be used for performance analysis such as identification of bottlenecks by looking at how much time each operation takes.

4 Discussion

In this section we outline strengths and weaknesses of SAVE, propose possible ways of improving the engine, discuss its applicability in different situations and do some preliminary performance analysis. It must be noted that the discussion is mostly bounded to the specifics of the "Win32/T-110.6220" virus family since other families may utilize totally different infection, obfuscation and operation schemes. However, modular design of SAVE allows to extend it and enable detection of other viruses.

One of the main rules of thumb in antivirus software development is to make the software work fast on clean and irrelevant files and do careful potentially slow analysis of suspicious files. Our engine follows this rule in two ways. First, it ignores files without a PE header, since

such files are non-executables and therefore can not be infected with the "Win32/T-110.6220" virus. Second, it dynamically detects presence of obfuscation and ignores the file if none is found. Thus, with an assumption that the virus infects only obfuscated files, absence of obfuscation greatly reduces processing time of clean files since the absence can be found very fast.

The current PE detection scheme is not optimal. Size of a PE header is not constant and varies from file to file depending on the number of sections, imported functions, etc. Currently SAVE is not able of dynamically identify the size of the PE header, therefore the whole file has to be passed to the `pefile` library we use to parse PE header. This is inefficient since it implies that the whole file is read into memory before PE header parsing. An optimal solution would be either to identify the size of the PE header and read only these bytes into memory or use another PE parser library that is capable of working with streams of data, reads the file byte by byte and stops when it finishes parsing the header.

Obfuscation detection scheme is based on the knowledge of how the "Win32/T-110.6220" obfuscation works - if the three parameters needed for deobfuscation are present, then the file has an obfuscated body. This algorithm makes several assumptions that may not hold for other obfuscation schemes or even for modifications of the "Win32/T-110.6220" scheme. First, it assumes that these parameters are initialized within first N instructions from the program entry point with N currently being equal to 20. Second, initialization is assumed to be done in a particular way, i.e. by using `mov esi, mov ecx` and `mov bl` instructions for storing obfuscated body address, size and key respectively. Moreover, detection of these instructions is done by simply looking for their byte opcodes (`b6`, `b9`, `b3`) without using any disassembler library. Thus if someone inserts a fake instruction having one of the above byte opcodes as an argument (e.g. `mov dx, 40be00`), SAVE will misinterpret the parameter initialization instruction. To make the scheme more robust a disassembler library which is able to correctly reconstruct opcodes and their arguments must be used. A more reliable way to do obfuscation detection is to base it not only on the presence of parameters, but also on the presence of deobfuscating loop which immediately follows parameters initialization phase. Finally, for some specific analysis tasks it may be helpful to extract the

'real' entry point by considering the jump instruction that follows the deobfuscation loop.

As stated in Section 3, the analysis module matches the file data against the fingerprint database. The imported function fingerprint test is fairly straightforward and there is little that can be done to improve it apart from using a more efficient PE header parser. However, the other two tests — string fingerprint tests — can be improved significantly. Currently the substring matching is done using Python's built-in `IN` operator which presumably implements a naïve algorithm with $O(m * (n - m + 1))$ complexity with respect to obfuscated body size n and fingerprint string length m . String matching performance can be greatly improved using advanced algorithms, such as Aho-Corasick, which has linear rather than higher order polynomial complexity with respect to the lengths of the matched string and the text. In addition, Aho-Corasick makes it possible to efficiently search for a set of strings instead of matching them one by one.

We did a simple performance analysis to identify potential bottlenecks of the engine. We found that in total each of the ten files is processed about 4 milliseconds. All the three main steps - PE header parsing, deobfuscation and analysis have about the same duration in the range 1 – 1.6 milliseconds. As we partially noted above, PE parsing performance depends on efficiency of the PE parser library; deobfuscation performance is linear in respect to the obfuscated body size and can be improved e.g. by stopping deobfuscation when the required fingerprint string is found, which is however impossible when several fingerprints are to be found; analysis complexity scales linearly in respect to the fingerprint database size and can be improved with better string matching algorithms. The above performance values constitute the lower bound since the file size, obfuscation body size and fingerprint database size can hardly be smaller. More elaborate analysis with bigger sample size and higher variety of files and conditions is needed in order to make more robust conclusions about bottlenecks of the engine.

There are also other minor factors that may affect performance of SAVE. The programming language used to implement it — Python — is an interpreted language and therefore inevitably has worse performance than compiled languages such as C/C++. The logger module currently is not efficient since every time it receives a log message, it immediately stores it to the file on disk. To avoid fre-

quent disk I/O operations it may be possible to implement a memory buffer for storing log strings which is flushed to disk after some number of incoming messages.

5 Conclusion

In this report we described Simple AntiVirus Engine, a tool developed to detect the "Win32/T-110.6220" virus family. Detection steps include PE header parsing, extracting obfuscation parameters, removing obfuscation and matching certain parts of the file against the virus fingerprint database.

The engine architecture is designed in a generic, flexible and modular way in order to allow easy extension of existing features. The analysis of engine's components shows that there is much space for future work that would help improve scalability, efficiency and robustness of the engine.