Journal of Theoretical and Applied Information Technology
10th March 2013. Vol. 49 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

ARCHITECTURE OF MALWARE DETECTOR FOR OBFUSCATED CODE INSPECTION

[1]LEE LING CHUAN, [2]MAHAMOD ISMAIL, [2]KASMIRAN JUMARI, [4]CHAN LEE YEE

[1,4]PhD Student, Department of Electrical, Electronic and System Engineering Nation University of Malaysia, Malaysia

[2,3]Professor, Department of Electrical, Electronic and System Engineering, National University of Malaysia, Malaysia

E-mail: [1]lclee_vx@f13-labs.net, [2]mahamod@eng.ukm.my, [3]kbj@eng.ukm.my , [4]chanleeyee@f13-labs.net

ABSTRACT

Signature-based malware detection is a very fundamental technique that detects malware by generating signatures. The detection however, is unable to detect obfuscated malware unless pre-generated signature is stored in the database. In this paper, we propose a combination of known packer detection, unpacking module, and heuristic scanning techniques to find and block a malicious program before it manages to be executed locally. Unpacking is the process of stripping packer layers and restoring the original contents. This module contains self-decryption script bodies that are devised to detect and extract the hidden-code bodies of obfuscated malware. Hence, the scanning process only deals with real malware body but not junk block or junk subroutine code. This paper also draws up the implementation and the evaluation of our virus scanning mechanisms. Finally, we present experimental results of our proposed techniques and the results show that our test set is highly accurate.

Keywords: *Malware Detector, Obfuscated, Unpacking, Emulator, Disassembler*

1. INTRODUCTION

The effort of continuously developing applications for computer systems and the Internet has been giving malware programs chances to propagate their malicious activities. Malware can infiltrate computers using various methods; for instance, hidden functionality in regular programs, attacks against known software vulnerabilities, drive-by-download from unsafe web sites and more. Much research has been done by antivirus researchers to provide better protection for computer systems and its applications. Unfortunately, the efforts did not stop the growing of malware; instead, the techniques became more sophisticated [1]. Typical antivirus techniques detect these sophisticated malware to create more attack pattern sets. However, the huge signatures have caused many computers to slow down significantly [2]. The computational resource consumption by security scanning software is dependent on the amount of scanning data and the size of the pattern set. If the security scanning tool is deployed to protect a busy server machine with a significant size of data involved, the required throughput performance might not be achieved.

The challenge of designing a malicious program is to design one with the capability of infecting a computer without the victim's consent. Currently, techniques such as packing, encrypting and obfuscation are the popular methods that malware authors use to hide the malware's malicious functions [3]. These viruses are known as obfuscated virus [4]. Obfuscated virus has evolved from simple encryption and compression to metamorphic virus [5] and polymorphic virus [6]. Metamorphic virus uses variant obfuscation techniques to create morphed copies of any base malware file. As opposed to metamorphic virus, polymorphic virus mutates or changes by generating many unique encryption methods for encryption. Both techniques help in avoiding the detection of signature based methods. In spite of the fact that different obfuscation techniques have been used to protect the malware instance's innards, most obfuscation algorithms are available from the Internet (for example UPX, ASPack, Armadillo). Ironically, many malware that appear today are repacked versions with common packers; however, they still manage to effectively evade the detection of Antivirus software [7].

Conceptually, heuristic scanner [8] is devised to detect new and unknown malware. With proper design of scanning algorithms, the detection of existing virus family variants is possible. Heuristic scanner is devised in a manner of either static or

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

dynamic [9]. Static heuristic scanner detects a malicious program based on an analysis of code structure. Dynamic heuristic scanner implements emulation to simulate CPU and memory activities to detect malicious operations while the malware program is executed on an emulated platform.

Our approach is to design a hybrid method that combines the known packer detector and removal with a heuristic virus scanning engine to accelerate virus scanning in computers. As mentioned earlier, most obfuscated techniques used by malware authors are from known packer. Dynamic heuristic scanner is capable in unpacking obfuscated executables in memory by executing the instance code on the virtual memory. The approach of known packer removal can accelerate the scanning process by detecting and removing any known packer starting from the common entry point and reveal the real intention of the malicious code instead of consuming computer time and performance to emulate and decrypt garbage instructions. In cases where no known packer is detected, the emulator component will be executed in virtual memory. This approach is based on the belief that no matter how complex the obfuscation algorithm is, the binary will eventually be decrypted in memory. Static heuristic scanner is devised based on an analysis which compares file format and an instance code fragment to a virus "pattern." The word "pattern" refers to the hexadecimal string in a virus signature. Our malicious behavior database is designed by using a sequence of one or more segments which are separated by gaps. Each time the scanning engine scans a malware instance file, the overall program's structure, computer instructions, programming logic and some other attributes will be scrutinized.

In summary, this paper demonstrates the capability of detection and removal of obfuscated techniques implemented by malware authors. We devised the packer detector approach based on signature to automate the process of identifying and extracting the hidden code bodies of packed executable files. The proposed method can accelerate the implementation of the malware detection processes. While the emulator is executed the obfuscation program in memory before the scanning and detection of malicious instructions is launched. Towards this end, we make several contributions; we proposed an approach of a malware signature database design that accelerates the process of malware detection. The signature database uses multiple parts of malware patterns to be matched in sequence for virus detection. This method can reduce the size of malware signature database and accelerate the process pattern matching by selecting a partial malware pattern to be matched instead of the whole full text of signature. We also proposed a design of a heuristic engine and emulator engine corresponding to a future threat that most malware detection software must deal with.

The rest of the paper is organized as follows. Background and related work is in Section 2. Section 3 describes the system architecture where the design of virus and packer signature database and the implementation of heuristic scanning will be explained in this section. The experimental results are discussed in Section 4. Thus, finally, conclusion is given in Section 5.

2. BACKGROUND AND RELATED WORK

This section briefly reviews the background and works related to this project. Although virus and malware detection has been studied for years, many modern malware programs are still evading existing malware detectors. Obfuscation is a common method that transforms the true purpose of the original program code into a misleading or unreadable form in hopes of hiding the program's true intentions. According to Brosch [10], more than 92% of malware files are runtime packed. In particular, malware obfuscation is the very first problem a malware analysis should be addressed. If an obfuscated malware instance cannot be unpacked, the analysis of the program will only view the obfuscated block as non-instruction data.

Malware detection can occur before, or after the malicious code is loaded into the memory. Thus, the detection approach can be categorized into static and dynamic strategies. Scott [11] presented a heuristic scanning method for detection of windows based obfuscated malware by scanning Windows PE structure before the binary is executed in memory. Sung [12] developed a robust signature based malware detection system; Static Analyzer of Vicious Executables (SAVE) which emphasizes on detection of obfuscated and mutated malware. The basic idea of this approach is that all versions of the same malware share a common core signature that is a combination of several features of the code. Schultz applied the Naïve Bayes's [13] method to detect previous unknown malicious codes. They designed a framework to train multiple classifiers on a set of malicious and benign executables to detect new viruses.

Journal of Theoretical and Applied Information Technology
10th March 2013. Vol. 49 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

In this paper, we focus on static heuristic scanning or the white-box approach in which the target malware instance is in hexadecimal format which enables our scanning approach to understand the whole code structure and functionality of the malware. The effectiveness of a virus scanning engine depends on the virus signature database. When a block of malware instance program is matched with a pattern set, the data file concerned is infected. To solve the problem of obfuscated malware executables using a great variety of packers [14] (for example, UPX, ASPack, Themida, NSPack.), we integrate the packer detector and packer unpacking module with our heuristic scanning engine.

3. THE SYSTEM ARCHITECTURE

As the name implies, the packer or code packing is an obfuscated technique that is used to hinder the true function of a binary program through reverse engineering. The intention of this technique, especially as applied by most malware authors, is to repackage the malicious program in ways that will alter the malware to make it appear completely different from the original binary; thus, effectively evading the detection by most malware detection software. In order to detect the variant malware which evolves from the implementation of packer or obfuscation techniques, antivirus companies would create a virus signature for each variant malware program. Due to this tendency, virus signatures increase significantly. In this paper, the proposed packer detection and de-obfuscation techniques would accelerate the overall scanning process and reduce the size of the virus signature database.

In this section, we describe the architecture of the malware detection system that is the core component of our malware and virus scanner framework. As shown in Figure 1, the core component consists of known packer detector, packer unpacking module, heuristic scanning engine and last but not least, the emulator and disassembler. Both known packer detector component and packer unpacking module component rely on the packer database. The packer database defines the packer signatures and the sequences of unpacking instructions. In case a known packer signature is detected by the targeted program, specific unpacking instructions will be executed to unpack the packed program to reveal the real functionality of the program. The virus signature database consists of the malicious signatures and its cure function. It is used by the heuristic scanning engine to apprehend if any malicious instructions are contained within the targeted program. The following section describes the functions of each core component.

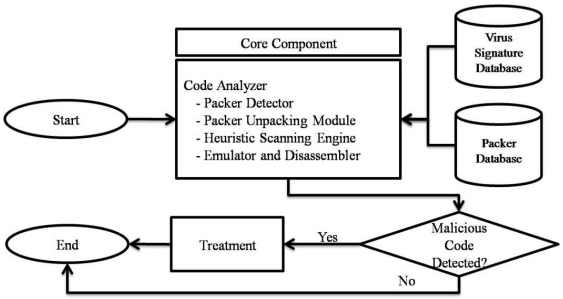

Figure 1. Functional Design of Malware Detector Core Engine

3.1 Packer Detector And Packer Unpacking Module

The approach of our detection engine is based on the natural behavior of the execution packer where the protection malware code will eventually be decrypted and revealed in memory, regardless the type of packer or the number of packed layers used. To defeat the obfuscation code implemented by the packer which poses obstacles to the virus scanner and detection engine, an automated process for identifying and extracting the hidden-code bodies is proposed.

In the packer detector component, we devised an algorithm that identifies whether a program applies any obfuscation mechanisms. Known packer detection function is built on top of core scanning components. It is developed to analyze a malware instance file, and determines if any packer has been applied. Our approach begins by detecting any packer applied for malware instance files based on the packer signature detection at entry point [15]. The entry point is the first instruction the pointer points to, which is intended as the destination of a long jump. A module for automating the process of extracting the hidden code to obtain the original code bodies of the program is executed if any packer is detected.

Figure 2 illustrates the architectural design of the packer signature database. As shown in the figure, seven entities are required to store the data for packer detection. The packer_no entity displays the amount of packer signatures inside the database. In this case, only one packer signature is available in the database. The remaining entities, _prefix_ftype, _prefix_fname, _prefix_signature_length, _prefix_reserved, _prefix_signature_data and _prefix_cure_offsetare defined with a serial of

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

prefix numbering identification. The _prefix_ftype determines if the targeted program is a Windows PE program. The name of the packer is defined under _prefix_fname. Whether a targeted program implements the packer is based on the signature matching process between the targeted binary with _prefix_signature_data. If the program perfectly matches, the execution pointer will jump to the particular unpacking module located at the offset address defined by _prefix_cure_offset.

```
.data                    ;signature database
packer_no       dd 1             ;number of records in the database
_001_ftype          db eftype_pe
_001_fname          db 'Pack.UPX.307'
                    db 013 dup (0)
_001_signature_length    db 011
_001_reserved       dw ?
_001_signature_data db 060h,0BEh, '*',004d,08Dh,0BEh,'*', 004d  ;ollydbg=60h,BEh,3Fh,04h,8Dh,BEh
                    db 057h,083h,0CDh
                    db 019 dup (0)
_001_cure_offset      dd 0
```

Figure 2. Packer Signature Database

3.1.1 Scanning and matching process

Generally, packer signatures can be defined as a set of instruction sequences that contain the most significant information to represent a particular packer. The scanning process to determine whether a file contains the obfuscation instruction relies on a matching process between the body code of the targeted program, P, and the packer signature, T. The targeted program will flag as packed by a particular packer if a match is found. In order to reduce the size of database and increase the effectiveness of the scanning process, the matching process works together with wildcard techniques in which skipping of bytes and byte ranges is allowed. In our framework, the wildcard character, "*", is defined to determine the number of character that are skipped between the two consecutive signature letters of the body code of a file.

```
Signature_data _001    db   0B4h, 03Ch, 0BBh, 000h, '*', 009d, 026h, 0FFh, 01Eh, 084h,
                            000h
```

Figure 3. Fragments of a Malware Signature

Figure 3 uses a wild card regular expression to divide the signature into two segments. As shown in the example, upon a hit of 0B4h, 03Ch, 0BBh, 000h,if 026h, 0FFh, 01Eh, 084h, 000h appears after the skip of 9 bytes distance from the first segment, only then it is possible to report the file as a packed instance. The value of an arbitrary amount of bytes after (*) wild card indicates the distance in bytes between two segments.

The pattern matching process of our scanning engine uses a variation of the Aho-Corasick pattern matching algorithm [16], which prepares for the on-going future plan that will consist of a large number of patterns against input text inside the database.

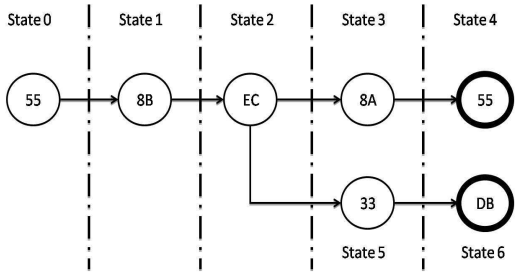

Figure 4. Success Transitions of keyword searching

The Aho-Corasick algorithm is initialized by building a finite state machine for the entire signature pattern with the purpose of constructing a pattern matching automaton. Figure 4 shows the automaton for the signature of "55 8B EC 8A 55" and "55 8B EC 33 DB". State 0 illustrates the beginning of the automaton, and both of the final states are shown in bold circles. The first signature pattern, "55 8B EC 8A 55" is added at state 0 until state 4. Since the second signature shares the same prefix ("55 8B EC") of the first signature, only state 5 and state 6 are needed to be created.

All signatures with the same prefix are stored in a linked list under the appropriate trie leaf node. As long as the trie is built, the pattern matching process is ready to read the opcode of the targeted program whether it matches with any of the patterns in the trie. If the match is confirmed, it will follow the trie transaction and check the entire pattern inside the linked list using a sequential string comparison method. The process proceeds until the last input opcode is read or a match failure is detected.

3.2 Heuristic Scanning Engine

The idea of heuristic scanning is to detect the most significant malware functionalities statically without executing the targeted program. Figure 5 illustrates the idea of our heuristic scanner engine. Unlike dynamic analysis, static extraction analysis provides complete information on targeted instances via PE parsing approach. Generally, the PE parse transforms Windows binary files to collect information for the purpose of pre-automating the analysis. The static extraction step for information

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

collection is quite complex and require several processing steps.

Static extraction, illustrated on the left side of Figure 5, uses the traditional extraction technique to collect necessary information. As shown in the figure, the initialization function displays the option of the scanning engine. The command line argument component collects the scanning option for the next instruction action. Prior to beginning the heuristic scanning process, the information of the scan target is crucial to prepare the drive path and search for file components. By calling Windows API functions, which include GetCurrentDirectory, FindFirstFile and FindNextFile,information such as the scanning path, names and total number of targeted scanning files can be collected. The last information collective step, the process file component, identifies the file permission, file type and size of each scanned program. As soon as sufficient information is collected, the information will be emitted to the heuristic scanning component to perform the matching process for identifying whether the targeted program is benign or malicious.

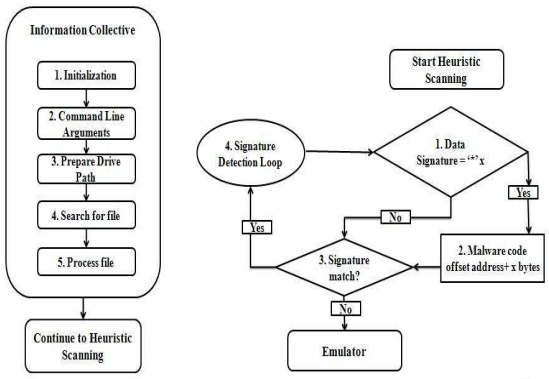

Figure 5. Flowchart To Describe The Overall Process Of Static Heuristic Scanning

The heuristic process, as shown on the right side of Figure 5 performs the operation according to the following steps:

Step 1: Data Signature='*'x.The process begins by detection of asterisk character (*) wild cards. Symbol (x) represents arbitrary bytes of value for a gap between two segments that was predefined by antivirus analysts. On the condition that the scanning pattern character is not equal to the asterisk character (*), it will jump to Step 3 to perform the comparison with thevirus signature database. On the contrary, if the scanning process matches the asterisk character (*) wild card, it will proceed to step 2.

Step 2: Malware Code Offset address+ x bytes.The pointer of the scanner will move to the next pattern segment with predefined length of gaps. The process will proceed to step 3.

Step 3: Signature Match. This stage performs the string pattern matching process with the signature database. Upon a hit of a signature matched, the process will jump to step 4 to prepare for the next scanning loop. However, if no match is reported, the heuristic scanning process will stop and the remaining incomplete scanned file will be passed to the emulator and disassembler module [17]. Emulator is a safe virtual environment in which to spot and trace the next instruction of an instance executable program.

Step 4: Signature Detection Loop.The scanning pattern pointer will shift to the next character and the scanning approach will reiterate from step 1.

3.2.1 Taxonomy of Virus Detection

In general, the heuristic scanning examines characteristics of the scanned target program code, which includes the file size, its architecture and behavior to determine the likelihood of an infection. It intends to duplicate expert antivirus analysis by looking for specific signatures with the likeliness of a virus or certain unusual instructions or commands which of these are not found in typical application programs. The heuristic scanning performs in a manner that uses a search and detect function to scan for pieces of hexadecimal code that are generally "viral-like" and do not have known signatures.

As mentioned earlier, heuristics scanning is a method that looks for "viral-like" activities. Unlike traditional signature detection, heuristic scanning involves static extraction and verification of either benign or malicious of an executable based on behavioral signature, not simple byte patterns. Behavioral signature is a program with distinct syntaxes that have identical malware behavior capture signatures. With the design of malware behavior signatures, the ability to detect a malware no longer relies on detecting a single piece of malware program but a whole class of malware from a common strain.

Figure 6 shows an example of pieces of code that perform actions in a way that we have specified as malicious. The left side of the figure shows the operational code (Opcode) [18]. It is part of a machine language instruction that specifies the

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

operation to be executed and it is readable by microprocessors. On the contrary, the Disassemble Code [19] on the right side displays assembly languages in human-readable format that have been translated from machine language. The example code in Figure 6 illustrates a program logic that performs a function which determines the entry point of a targeted program and returns to the normal execution flow. During infection, a malware program does not know the exact address in memory until the allocation is made. Moreover, the allocation address might be different from an execution to another. This is a common automaton virus infection mechanism to retrieve the memory address of the entry point for overwriting or moving programs in memory.

	Opcode	Disassemble Code
1.	B4 3C	MOV AH, 3C
2.	BB 00 00	MOV BX, 0000
3.	5D 81ED <offset address>	POP EBP SUB EBP, <offset address>
4.	66 813E 5045	CMP WORD PTR[ESI], 'PE'
5.	9C	PUSHF
6.	66 813E 4D5A	CMP WORD PTR[ESI], 'ZM'
7.	66 813E 4B45524E	CMP DWORD PTR[ESI], 'KERN'

Figure 6. Sample piece of malicious code found on a malicious executable file

Our malware and virus scanner detection engine approach comprises of a scanning engine module and a malware signature database. Both modules work together and are inseparable. Generally, the design of our signature database is highly volatile. The main goal of volatility is to ensure new signatures can be updated in the future.

```
@005_ftype        db eftype_pe
@005_fname        db 'Virus.Win32.Funlove.4070'
                  db 001 dup (0)
@005_sig_len      db 015
@005_reserved     dw ?
@005_sig_data     db 0E8h,0AAh,008h,000h,000h,08Dh, '*',005d,08Bh,03Ch
                  db 024h, '*',006d,0A5h,0A5h
                  db 15 dup (0)
@005_cure_offset  dd offset funlove
```

Figure 7. Virus Signature Database

The design of our detection engine to detect both malicious code and known packer for instance executable files is similar; both using a signature database. Figure 7 shows the architectural design of a virus sample and packer signature databases, respectively. Similar to the architecture of the packer signature database shown in Figure 2, all entities will be defined with a serial of prefix numbering identification. As shown in Figure 7, consider the fifth group of virus sample signature, @005_ftype and @005_fname, which represents the type of executable file and name of the malware instance, respectively. eftype_pe represents the PE file format. @005_sig_len specifies total length of the signature. In addition, the signatures were stored in the most efficient Opcode data type, as shown in @005_sig_data. The selection of the signature is based on the significant behavior of funlove. The virus implements the modification PE structure method by inserting its malicious code in the .reloc table in PE structure.This behavior would never happen for a normal benign program. Our approach of @005_cure_offset will trigger the scanner to proceed to the funlove cure function if the infection of Virus.Win32.FunLove.4070 was detected. @005_reserved takes no action and is reserved for future usage.

3.3 Emulator and Disassembler

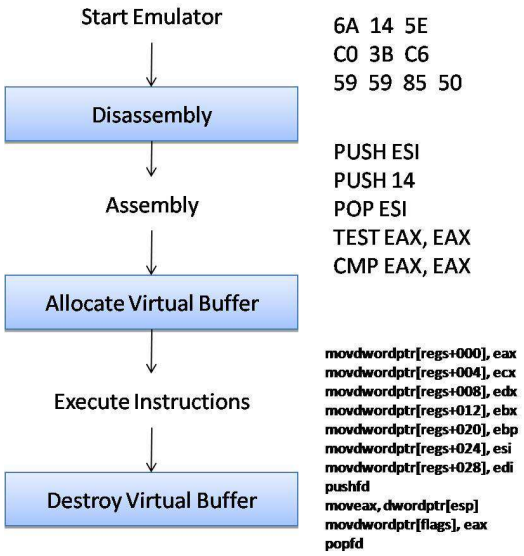

Figure 8. Incremental steps of the Emulator and Disassembler

The emulation identifies common malicious activities via emulating the instruction codes of a targeted program. Thus, a safe and isolated environment is crucial to perform a just-in-time binary execution within the environment to prevent the execution of malicious instructions that can cause damage to the local computer. To emulate every instruction, the CPU emulation is devised to become the core of the emulator engine. Figure 8 illustrates the overall steps of the emulator engine. The process flow begins with the Disassembly

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

component translating the targeted program from opcode into assembly instructions. After the disassembly process, a safe virtual environment is ready to allow the translated instructions to be executed.

A safe virtual environment requires a list of virtual CPU register for support when performing the corresponding instructions. The execution of the target sequence will call the defined virtual CPU without access to the original register. During execution, the virtual CPU has to check whether an existing block of instructions consists of malicious code. The virtual buffer of the emulator would be destroyed if any malware signature is detected or the maximum allowed time for analysis has elapsed. All original register saved on the stack must be destroyed before handling a pointer to conclusion, where it will be decided whether the scanning program was infected by a malware or not.

4. IMPLEMENTATION AND EXPERIMENTAL RESULTS

4.1 Implementation And Experimental Results

The implementation of known packer and virus signature detection described in this section is fairly conventional. Heuristic scanning engine uses a specific packer signature database to determine if any packed-code is applied by an instance binary. If a packed-signature match is detected, our unpacking mechanism is used to unpack and extract the hidden code contained in the target binary file. A specific virus signature database is used to determine if any malicious program exists inside an instance binary. The virus signature database refers to common short signatures, which are presented in most viruses (also known as "suspicious" command). Our approach is to select different code segments from a common short signature and save it into our virus signature database. If a match is found, the instance file is flagged as virus.

Figure 9 illustrates the packer detection of the obfuscation mechanism applied by Worm.Win32.QAZ [20] together with a predefined packer signature database. As shown in the figure, the upper part of Worm.Win32.QAZ packed with UPX packer refers to the hexadecimal format [21] of the packed binary. It is a hexadecimal view of malware binary and each byte is represented as a two-digit hexadecimal number. Parameter is a pre-allocated variable that exists in the x86 registers [18] before the heuristic scanning process starts. Conceptually, many instructions assign specific registers of certain arguments. For instance, string instructions use ECX as a size of signature, ESI as a source pointer of signature database and EDI as a pointer to the first byte of malware binary at entry point. Thus, the three variables; signature length, signature data and code location, are assigned to ECX, ESI and EDI respectively.

Packer signature database illustrates the design of packer database. packer_no shows that there is only one packer signature inside our database. Ultimate Packer for eXecutables (UPX) [22] is the obfuscated mechanism used by the instance malware defined by @001_fname entity. @001_sig_len indicates the total length of packer signature. Before the next heuristic scanning process begins, the target instance file stays in an unpack form. Thus, any detection of known packer reported, the detection mechanism will trigger an automatic unpacking process to reveal the innards of the instance binary file.

As shown in Figure 9, a total of eleven bytes of length has been defined, and the ECX register will inherit the value for future scanning processes. The heuristic scanning engine, fully coded in assembly language, shows the overall scanning process. According to the program, sig_detec_loop is designed to detect any asterisk character (*), while the other two functions, detection_compare and detection_cont are designed to perform the scanning/pattern matching process and position/shift values corresponding to possible blocks respectively.

Prior to starting our packer detection codes, the total length of the generated signature pattern, @001_sig_len is saved in the ECX register. The value is of eleven bytes including asterisk character (*). As shown in Figure 9, the heuristic scanning process will begin at the sig_detec_loop function to detect any asterisk character (*). If none of the asterisk character is detected, the scanning process will jump to the detection_compare function to perform the comparison of instance binary with the packer signature database. Upon a hit of 060h, 0BEh opcodes, the sig_detec_loop function is able to detect an asterisk character (*) and the scanning process jumps to the detection_cont function. The counter value of ECX register will decrease and the pointer of the EDI register will move to the next pattern segment with a predefined length of gaps. In this case, the total predefined value of gap is 4 bytes. The scanning process will shift to the next character and the scanning approach will loop back to the sig_detec_loop function. The scanning process will reiterate until the value of ECX counter becomes zero or the match of the entire signature at @001_sig_data is reported.

Journal of Theoretical and Applied Information Technology
10th March 2013. Vol. 49 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

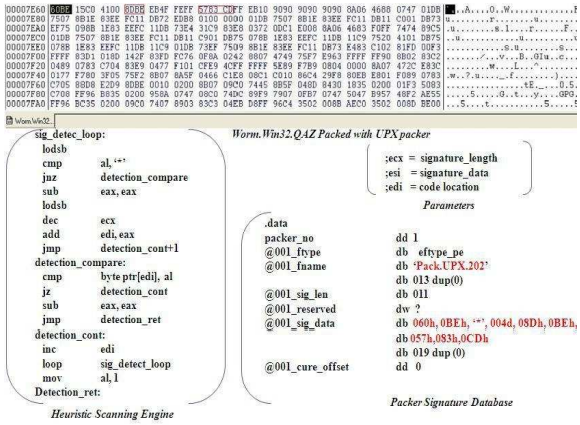

Figure 9. Packer Detection of Heuristic Scanning Engine

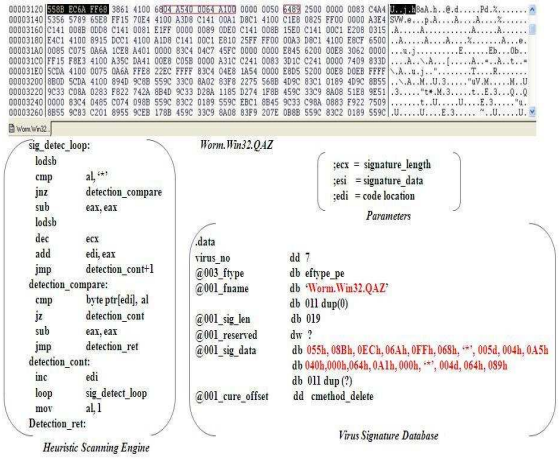

Figure 10. Virus Detection of Heuristic Scanning Engine

The process of detecting Worm.Win32.QAZ virus illustrated in Figure10 shows that the pattern matching process between the malware binary with the virus signature occurred at the detection_compare function. After a hit of series 055h 08Bh 0ECh 06Ah 0FFh 068h, the sig_detec_loop function succeeds in detecting asterisk character (*) and the overall process jumps to detection_cont. The counter value in ECX register decreases and the pointer of EDI register moves to the next pattern segment with predefined length of gaps which is five bytes. The scanning process shifts to the next character and the scanning approach continues to the sig_detec_loop function. The scanning process repeats until the value of ECX counter becomes zero or the match of the entire signature at @001_sig_data is reported.

4.2 Experimental Analysis

The experiment began with the detection of real malware from the Internet which were once infamous causing millions of dollars lost at its appearance. Seven viruses including Marburg [23], FunLove [24], Kriz.4029 [25], Parite.B [26], Worm.QAZ [27], Confickervarian B and Confickervarian C [28] have been collected from a virus collectionwebsite, VX heaven [29]. Nevertheless, the website is currently seized by local police forces due to the criminal investigation in regards to the articles of 361-1, “Criminal Code of Ukraine – The creation of malicious program with an intent to sell or spread them”, based on someone's tip-off on “Placement into the free access malicious software designed for the unauthorized breaking into computers, automated systems, computer networks” [30].

As soon as the targeted viruses were collected from the VX Heaven website, the experimental detection was initialized via detecting the targeted viruses with the proposed malware scanner engine. The malware detection test is built based on the malware samples that have been collected. Table 1 illustrates the proposed malware scanner and detection results. Under the detection column, only two options are allowed, that is either √ or X. √ indicates detection, X indicates failure to detect. As shown in Table 1, the seven malware are detected by the proposed malware scanner engine.

Virus.Win9x.Marburg.b was the virus that used real 32-bit polymorphic engines. It can create endless numbers of new decryptors via different encryption methods to encrypt the constant part of the virus body. Our detection malware tool, constructed by both dynamic decryption and emulator, has the ability to deal with this obfuscation technique. As mentioned earlier, the intention of the packer unpacking module is to unpack the known detectable packer; however, the number of unpacking algorithm in the database is limited. In order to defeat the new and undetectable packer, the emulator component is proposed to the scanner engine. Table 2 examines the detectable time and cure time of the emulator component. In this test, the unpacking algorithm of the viruses;Virus.Win9x.Marburg.b, Conficker variant B and Conficker variant C, are not found in our database. The emulator component will be triggered to unpack the obfuscation portion to reveal the malicious code to the scanner to detect the abnormal instructions. Table 2 shows the scanning time of the emulator.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 1: Feature Analysis for detecting virus and malware

Virus / Malware (MD5)	Detection (√ or X)	File Size (KB)
Virus.Win9x.Marburg.b (e8e0f1f5305718a03432a09fe38ab007)	√	28
Virus.Win32.Kriz.4029 (79c5d6806145b67968528ffe806990c0)	√	470
Virus.Win32.Funlove.4070 (fe05b8bb9eabcdafe0125334d02cb65a)	√	61
Worm.Win32.QAZ (1e9307bc19a0a7270c501c5e9108d214)	√	118
Virus.Win32.Parite.b (f689a4564a3bdb3f62093ab10e713180)	√	338
Conficker Variant B (6ee741c4e0d36d0dc9162a6e71943379)	√	158
Conficker Variant C (5e279ef7fcb58f841199e0ff55cdea8b)	√	86.5

Table 2: Detection and Recovery Time

Virus / Malware	Emulation Detection Time (Milli-seconds)	Emulation Detection and Cure Time (Milli-seconds)	File Size (KB)
Virus.Win9x.Marburg.b (e8e0f1f5305718a03432a09fe38ab007)	47	63	28
Virus.Win32.Kriz.4029 (79c5d6806145b67968528ffe806990c0)	125	140	470
Virus.Win32.Funlove.4070 (fe05b8bb9eabcdafe0125334d02cb65a)	16	18	61
Worm.Win32.QAZ (1e9307bc19a0a7270c501c5e9108d214)	16	17	118
Virus.Win32.Parite.b (f689a4564a3bdb3f62093ab10e713180)	31	32	338
Conficker Variant B (6ee741c4e0d36d0dc9162a6e71943379)	16	17	158
Conficker Variant C (5e279ef7fcb58f841199e0ff55cdea8b)	15	17	86.5

The detection, without any resistance techniques, lacks effectiveness in the scanning and detection engine. Thus, the experimental task is followed by implementationof The Ultimate Packer for eXecutables (UPX) packer with the viruses, and the same previous detection task is conducted. Table 3 shows the detection of obfuscation viruses and malware with the proposed scanner engine. Only 3 viruses are packed for the examination which includes Virus.Win32.Kriz.4029, Virus.Win32.Funlove.4070 and Worm.Win32.QAZ.

Table 3: Feature Analysis for detecting virus and malware

Virus / Malware with UPX Packer (MD5)	Detection (√ or X)	File Size (KB)
Virus.Win32.Kriz.4029 (a583bd8e66c4afeb5b27d28c51f3153e)	√	186
Virus.Win32.Funlove.4070 (a583bd8e66c4afeb5b27d28c51f3153e)	√	24
Worm.Win32.QAZ (57d0fe9f2c1531bd87c08af6ddd74bd7)	√	34.6

Table 4 shows the required time of the proposed scanner engine to remove the obfuscated packer and detect the malicious code. While comparing the results with Table 2, the file sizes of the samples are significantly reduced. Both removal time and curing time are almost similar to the original sample from Table 2. Summarizing, the return performance results are good where the required time to unpack and cure the malicious portion is less than one second.

Table 4: Detection and Recovery Time

Virus / Malware with UPX Packer (MD5)	Remove Packer and Detection Time (Milli-seconds)	Remove Packer and Cure Time (Milli-seconds)	File Size (KB)
Virus.Win32.Kriz.4029 (a583bd8e66c4afeb5b27d28c51f3153e)	130	140	186
Virus.Win32.Funlove.4070 (a583bd8e66c4afeb5b27d28c51f3153e)	17	19	24
Worm.Win32.QAZ (57d0fe9f2c1531bd87c08af6ddd74bd7)	17	18	34.6

4.3 The interface of malware detection engine

In this section, the final result of malware detection engine will be discussed. Our malware detection engine can be performed in the form of console and graphical user interface based. In order to maintain the light weight and reduce the scanning time, the entire engine is built using assembly language. However, Python programming is integrated with the assembly language to develop the graphical user interface of the malware detection engine. The main purpose of the graphical user interface is to reduce complications and increase the flexibility for users to execute the scanner engine.

Journal of Theoretical and Applied Information Technology
10th March 2013. Vol. 49 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

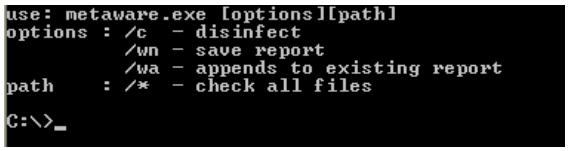

Figure 11: The Option of Execution of Malware Detection Engine

Figure 11 shows the execution options of the detection engine in console based. As shown in the figure, the options are "/c", "/wn" and "wa". The "disinfect" option ("/c") is designed to detect and delete the malicious instruction of a targeted executable program. Without this option, the engine will only perform the detection process. The option "save report" ("/wn") performs the scanning results and allows users to review the output of the scan. Last but not least, the option "appends to existing report" ("/wa") is to append the existing report in the same result file. The "[path]" allows the user to select the desired directory. The user can specify the desired path by inputting the directory of the path (e.g. c:\, c:\Document and Settings\User\Desktop etc.). However, the user can also perform a full scan of the entire directory in the user's computer. This can be done by inputting the "/*".

Figure 12 shows the graphical user interface of the malware scanner engine that was proposed in this paper. The graphical user interface approach allows the user to conduct a scan without having to input any options which may cause confusion. As soon as the scan reaches the end, another window will appear displaying the scan results as shown in Figure 13.

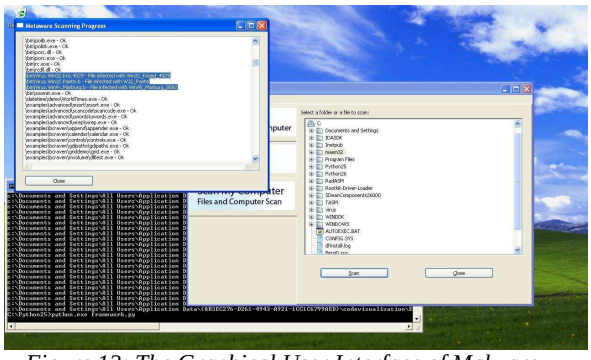

Figure 12: The Graphical User Interface of Malware Scanner Engine

In order to validate the scan engine, three real time malware were randomly inserted into a testing computer. The malware scanning core engine begins by scanning the entire C directory. As shown in Figure 13, a new window will pop up after the scanning process which displays that the scan has detected three executable files infected by malware out of 290 executable files that were scanned.

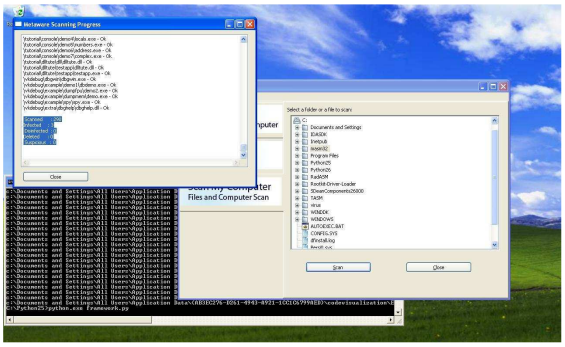

Figure 13: Results from Malware Scanning Core Engine

5. CONCLUSION

The analysis and detection of malware has beena time-consuming and challenging task. In this paper, we presented a heuristic method for detection of obfuscated or mutated window based malware. Our method scans for suspicious behavior patterns in the malware instance file before the binary is executed locally. In order to defeat the obfuscated techniques which areapplied by most malware, an automated process for identifying and extracting the original code bodies is proposed. This method prevents the scanning process from consuming computer resources by scanning junk blocks or junk subroutine codes. Our virus detection approach is based on the combination of packer detection, packer removal, and heuristic scanning concept. In this paper, the overall concept and algorithm of the core component have been described.

ACKNOWLEDGMENT:

The authors gratefully acknowledge that financial support for this research was received from a grant funded by Universiti Kebangsaan Malaysia under grant number oup-2012-182

REFRENCES:

[1] Gupta, A., et al. An empirical study of malware evolution. in Proceedings of the First international conference on COMmunication Systems And NETworks. 2009. Bangalore, India

[2] Clementi, A. Release rates & update size of signature databases of some main top Anti-virus products. 2006; Available from: http://www.av-comparatives.org/seiten/ergebnisse/Release_rates.pdf.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

[3] Kang, M.G., P. Poosankam, and H. Yin, Renovo: a hidden code extractor for packed executables, in Proceedings of the 2007 ACM workshop on Recurring malcode2007, ACM: Alexandria, Virginia, USA. p. 46-53.

[4] Priyadarshi, S., Metamorphic Detection via Emulation, in The Faculty of the Department of Computer Science2011, San Jose State University. p. 84.

[5] Chouchane, M.R. and A. Lakhotia, Using engine signature to detect metamorphic malware, in Proceedings of the 4th ACM workshop on Recurring malcode2006, ACM: Alexandria, Virginia, USA. p. 73-78.

[6] Abdulla, S.M. and O. Zakaria, Devising a Biological Model to Detect Polymorphic Computer Viruses Artificial Immune System (AIM): Review, in Proceedings of the 2009 International Conference on Computer Technology and Development - Volume 012009, IEEE Computer Society. p. 300-304.

[7] Desai, P., Towards an undetectable computer virus, in The Faculty of the Department of Computer Science2008, San Jose State University.

[8] Pao, D., et al., String searching Engine for virus scanning. IEEE Trans. Computers, 2011. 60(11): p. 1596-1609.

[9] Symantec Understanding Heuristics: Symantec's Bloodhound Technology. XXXIV.

[10] Brosch, T. and M. Morgenstern. Runtime Packers: The Hidden Problem? in Black Hat USA 2006. 2006. Las Vegas, USA.

[11] Treadwell, S. and M. Zhou. A heuristic approach for detection of obfuscated malware. in Proceedings of the 2009 IEEE international conference on Intelligence and security informatics. 2009. Richardson, Texas, USA.

[12] Sung, A.H., et al., Static Analyzer of Vicious Executables (SAVE), in Proceedings of the 20th Annual Computer Security Applications Conference2004, IEEE Computer Society. p. 326-334.

[13] Schultz, M.G., et al., Data Mining Methods for Detection of New Malicious Executables, in Proceedings of the 2001 IEEE Symposium on Security and Privacy2001, IEEE Computer Society. p. 38.

[14] Wu, Y., T.-C. Chiueh, and C. Zhao, Efficient and Automatic Instrumentation for Packed Binaries, in Proceedings of the 3rd International Conference and Workshops on Advances in Information Security and Assurance2009, Springer-Verlag: Seoul, Korea. p. 307-316.

[15] Eagle, C., The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler 2011 San Franscisco: William Pollock. 672.

[16] Aho, A.V. and M.J. Corasick, Efficient string matching: an aid to bibliographic search. Commun. ACM, 1975. 18(6): p. 333-340.

[17] Ligh, M., et al., Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code2010: Wiley.

[18] Hyde, R., The Art of Assembly Language2003: No Starch Press. 1000.

[19] Ferguson, J. and D. Kaminsky, Reverse Engineering Code With IDA Pro2008: Syngress Pub.

[20] Marx, A., The Usual Suspects - Part 3, in Virus Bulletin February 20012001, Virus Bulletin Ltd. p. 14-16.

[21] Yan, W. and E. Wu. Toward Automatic Discovery of Malware Signature for Anti-virus Cloud Computing. 2009; Available from: http://us.trendmicro.com/imperia/md/content/us/pdf/threats/securitylibrary/weiyan-09-whitepaper.pdf.

[22] Oberhumer, M.F.X.J. and L. Molnar. UPX: The Ultimate Packer for Executables. 2010; Available from: http://upx.sourceforge.net.

[23] F-Secure. 1998; Available from: http://www.f-secure.com/v-descs/marburg.shtml

[24] Kaspersky. How to disinfect computer from the virus Win32.FunLove? ; Available from: http://support2.kaspersky.com/38

[25] Panda. Virus Encyclopedia. [cited 2007 March 26]; Available from: http://www.pandasecurity.com/homeusers/security-info/24779/information/Kriz.4029

[26] Panda. Virus Encyclopedia. 2008 [cited 2008 December 12]; Available from: http://www.pandasecurity.com/homeusers/security-info/18181/Parite.B/.

[27] Kit, F.A. QAZ worm. 2000 [cited 2007 January 4]; Available from: http://www.fireav.com/virusinfo/library/qaz.htm

[28] Chuan, L.L., et al. Automating uncompressing and static analysis of Conficker worm. in Communications (MICC), 2009 IEEE 9th Malaysia International Conference. 2009. Kuala Lumpur.

[29] VXHeave. [cited 2012 April 20].

[30] CRIMINAL CODE OF UKRAINE. 2001; Available from: http://legislationline.org/documents/action/popup/id/16257/preview.