Towards Understanding Malware Behaviour by the Extraction of API Calls

Conference Paper · July 2010

DOI: 10.1109/CTC.2010.8

3 authors:

Mamoun Alazab
Australian National University
53 PUBLICATIONS **439** CITATIONS
SEE PROFILE

Sitalakshmi Venkatraman
Melbourne Polytechnic
78 PUBLICATIONS **603** CITATIONS
SEE PROFILE

Paul Watters
La Trobe University
156 PUBLICATIONS **1,434** CITATIONS
SEE PROFILE

Towards Understanding Malware Behaviour by the Extraction of API calls

Mamoun Alazab
Internet Commerce Security Laboratory (ICSL)
University of Ballarat
m.alazab@ballarat.edu.au

Sitalakshmi Venkataraman
Internet Commerce Security Laboratory (ICSL)
University of Ballarat
s.venkatraman@ballarat.edu.au

Paul Watters
Internet Commerce Security Laboratory (ICSL)
University of Ballarat
p.watters@ballarat.edu.au

Abstract— **One of the recent trends adopted by malware authors is to use packers or software tools that instigate code obfuscation in order to evade detection by antivirus scanners. With evasion techniques such as polymorphism and metamorphism malware is able to fool current detection techniques. Thus, security researchers and the anti-virus industry are facing a herculean task in extracting payloads hidden within packed executables. It is a common practice to use manual unpacking or static unpacking using some software tools and analyse the application programming interface (API) calls for malware detection. However, extracting these features from the unpacked executables for reverse obfuscation is labour intensive and requires deep knowledge of low-level programming that includes kernel and assembly language. This paper presents an automated method of extracting API call features and analysing them in order to understand their use for malicious purpose. While some research has been conducted in arriving at file birthmarks using API call features and the like, there is a scarcity of work that relates to features in malcodes. To address this gap, we attempt to automatically analyse and classify the behavior of API function calls based on the malicious intent hidden within any packed program. This paper uses four-step methodology for developing a fully automated system to arrive at six main categories of suspicious behavior of API call features.**

Keywords—Malware, API calls, code obfuscation, feature extraction.

I. INTRODUCTION

Malware has numerous synonyms such as, malicious software, malicious code (MC) and malcode. Malware contains code designed to perform illegal activities, cause damage, and affect the integrity and the functionality of digital electronic devices. For the purpose of this research, we adopt the malware description given by McGraw and Morrisett [1] as "any code added, changed, or removed from a software system in order to intentionally cause harm or subvert the intended function of the system". Sophistication in malware offers a new class of criminal activity that has created new challenges for law and forensic examiners [2]. Literature studies [3] [4] [5] on malware detection have shown that not all types of malware can be detected by one single technique. There two main techniques used for malware detection, namely, signature-based detection and anomaly-based (behavioural) detection. Anti-malware engines use signatures or 'byte sequences' to detect known malware. These signatures are generated by human experts by disassembling the file and selecting pieces of unique code. Signature-based detection is very effective for known malware, but the noteworthy weakness is the incapability to detect anonymous malware and hence is not effective against "zero day attack" (unknown malware) [6]. Code obfuscation has created another challenge for digital forensic examiners, namely the detection rate of new and unknown malware that is currently only being detected between 70 to 80% [7] [8] [9] and identifying benign code as malicious (false alarm rate or false positive) is quite high [7]. Signature based detection suffers from two hindrances, first, high false positive [7] (identifying benign files as malware) and second, high false negative (fail to detect malware) [8]. Therefore, in this research we focus on anomaly based detection using feature extraction such as application programming interface (API) calls and code obfuscation features. The obfuscation features include behaviours based on the content of malware such as source address, destination address, ASCII, UNICODE, and other contents that are relevant for the analysis [3] [5].

The Portable Executable (PE) is divided into sections. Each section provides different information, such as the file header, the number of DLL and the number of API calls. However, the information in the PE file header could be modified easily, and it has some difference from the true calls of program [34]. The existing techniques and methods do not perform sufficient statistical analyses to determine if the anomaly was 'actually' malicious. Therefore, in this research, we have used static anomaly-based detection analysis which consists of an introspection of the program codes to determine various dynamic properties of these codes in an isolated environment. We apply this technique to extract and identify the API function calls used by malware for 386 file samples that are uniquely named according to their MD5 value. We have obtained recent malware samples that were collected between July 2009 and November 2009 from honeypots, the Honeynet project and other sources. Our approach is static analysis of the malware based on the Windows API calling sequence and how to extract those windows calls that reflect the behaviour of a file.

978-0-7695-4186-0/10 $26.00 © 2010 IEEE
DOI 10.1109/CTC.2010.8

The paper is organized as follows: the next section highlights the main contributions of the study, in section 3 we describe the background of this research, which is based on a thorough foundation on API calls and PE file formats, and section 4 provides the methodology adopted and the system architecture of the fully-automated system implemented for this research study. We then discuss in section 5, the experimental results. Finally, we provide the limitations and future work of this ongoing research in section 6, and conclusions in section 7.

II. Contributions of the Paper

Recently, API calls have been explored for modeling program behaviour. There are studies [10] [11] [12] [35] that have used analysis of API calls for generation of birthmark on Portable execution (PE). Use of statistical analysis of file binary content including statistical N-gram modeling techniques [13] [14] [36] have been tested in identifying malcode in document files and does not have sufficient resolution to represent all class of file types. From our study on related work [15] [16] [17] [18] [19] [20] [21], we find that the statistical modeling of hidden malcode that predominantly use Windows API calling sequence for evading detection is yet to be explored. This is a motivation for this research towards a positive contribution in understanding malware behavior through statistical analyses of API calls.

In this paper, using the static analysis tool IDAPro disassembler [29], we disassemble, analyze and extract the API function calls from the binary content of malware, and statically identify the behavior from the API calls. We present a novel approach to automate and extract the API function calls from the malware binary content. We have applied a static anomaly based detection technique that consists of examining the malware programs without it being executed, so as to determine the behavior of the actual execution, and have combined it with API call feature extraction for reflecting the overall behavior of a file.

III. Background

Understanding the API call features and the format of program executables (PE) is of critical importance for identifying the hidden malcodes. Hence, some background information pertaining to API calls and PE structure that are exploited by malwares are presented here.

A. API Call Features

The core of the Windows O/S consists of the application programming interface (API). Windows API enables the programs to exploit the power of Windows and hence malware authors make use of the API calls as a vehicle to perform malicious actions [9]. The Windows API function calls comprises of functional levels such as system services, user interfaces, network resources, windows shell and libraries, etc. Since the API calls reflect the functional levels of a program, analysis of the API calls would lead to an understanding of the behaviour of the file. Malicious code are able to disguise their behaviour by using API functions provided under Win32 environment to implement their tasks. Therefore, in binary static analysis we focus on identifying all Windows API call features to understand the malware behaviour.

TABLE I
PORTABLE EXECUTION STRUCTURE

```
Section Table
DOS Header
COFF File Header
Optional Header
Standard fields
NT additional fields
Optional Header Data Directories
     Export Table
     Import Table
     Resource Table
     Exception Table
     Certificate Table
     Base Relocation Table
     Debug
     Architecture
     Global Ptr
     TLS Table
     Load Config Table
     Bound Import
     Import Address Table (IAT)
     Delay Import Descriptor
     COM+ Runtime Header
     Reserved
Section Table
     .text
     .rdata
     .data
     .idata
     .rsrc
```

In the Windows O/S, user applications rely on the interface provided within a set of libraries, such as kernel32.dll, ntdll.dll and user32.dll in order to access system resources including files, processes, network information and the registry. This interface is known as the Win32 API. Applications may also call functions in ntdll.dll known as the Native API. The Native API functions perform system calls in order to have the kernel provide the requested service. Our approach extracts and analyses these API call features including hooking of the system services that are responsible to manage files. The extracted calls are confined to those that affect the files. Various features related to the calls that create or modify files or even get information from the file to change some value and information about the DLLs loaded by the malware before the actual execution are considered. We then perform statistical testing on the extracted features to determine the malware class based on suspicious behaviours.

B. Malware Executable File Format

The Win32 Portable Executable (PE) file format introduced by Microsoft is the standard executable format for all versions of the operating systems on all supported processors. Therefore, our approach is tested directly on the

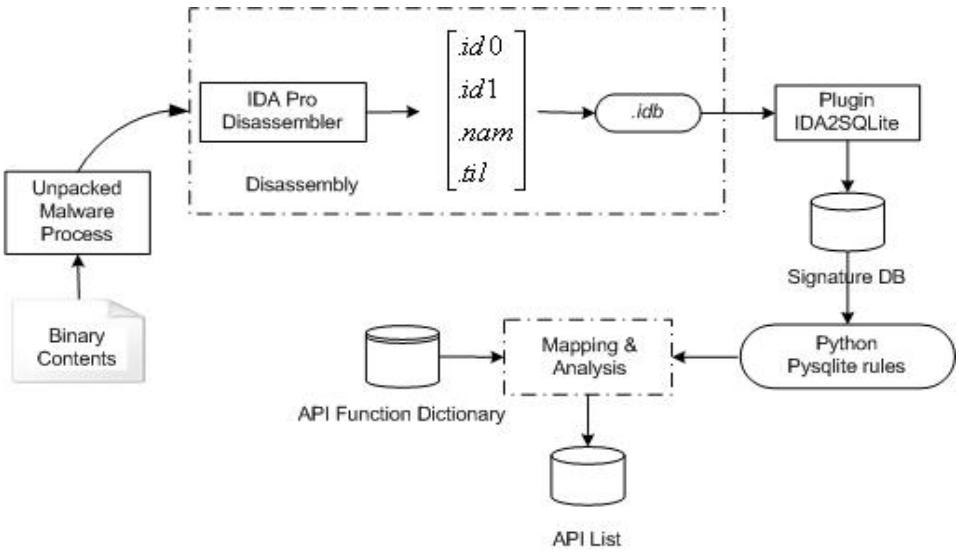

Figure 1 A fully-automated architecture for API call extraction and analysis of malware behaviour

PE format files. Most programs in windows are constructed by accessing the Windows API through functions available in dynamic link library (DLL) on the system. Microsoft provides a great number of DLLs, and each DLL can be used by more than one program at the same time.

For the obfuscated malware detection system, this paper focuses on extraction and analysis of features from API call sequences and how to automate the entire detection process. In order automate the inspection of these features in the PE, it is important to understand the PE structure as the data structures on secondary memory (Hard Disk) are the same data structures that are used dynamically in main memory (RAM). For example, when the function LoadLibrary is called to load the executable into memory, a data structure such as the IMAGE_NT_HEADERS is created identical on disk and in the memory. However, the Windows loader is responsible for deciding what section the PE need to be mapped in. Hence, the mapping between the two memory is consistent, such as the higher offsets in the file in disk is similar in position to higher memory addresses when mapped into main memory. Since a PE file have a different sections and headers, we have used the free PE tools [22] to view and analyse the PE files. Table 1 provides a good understanding of the PE structure that includes DOS headers and PE headers. The PE header start with the signature of the PE, file properties, such as the number of sections and timestamp, as shown in Table 1. The section table has code sections (.text), and data sections (.data). The .text section is the default section for code and the .data section stores writable global variables and also contains the file's Original Entry Point (OEP) which refers to the execution entry point (where the file execution begins) of a portable executable file. Finally, the .rdata section contains read-only data.

Traditional detection engines search for signatures to detect and identify known malwares. However, the malware authors can fool the detection engine by applying obfuscation technique using packers or polymorphism to safeguard the malware code and its data structures from being detected.

IV. Methodology and Implementation

In this section we describe the methodology adopted for automating the API extraction, analysis and malicious behavior identification process. Figure 1 shows the system architecture of such an automated process. We used Python programming language to implement the following main steps of the fully automated system:

Step 1: Unpack the malware.
Step 2: Disassemble the binary executable to retrieve the assembly program.
Step 3: Extract API calls and important machine-code features from the disassembly program.
Step 4: Map the API calls with MSDN library and analyse the malicious behaviour.

Step 1: Unpack the malware

Packers are commonly used today for code obfuscation or compression. Packers are software programs that are able to be used to compress and encrypt the PE in secondary memory and restore the original executable image, when loaded into main memory (RAM) [9]. Recently, malware authors have used packers to avoid detection and to run malware faster. Packing the malware makes the obfuscation method difficult to understand and the malware authors only need to change a small number of lines of code in order to change the malware signature. This is mainly as changing any byte sequence in the PE results in a new different byte sequence in the new produced packed PE [28] [37]. Malware authors are continually developing new techniques for

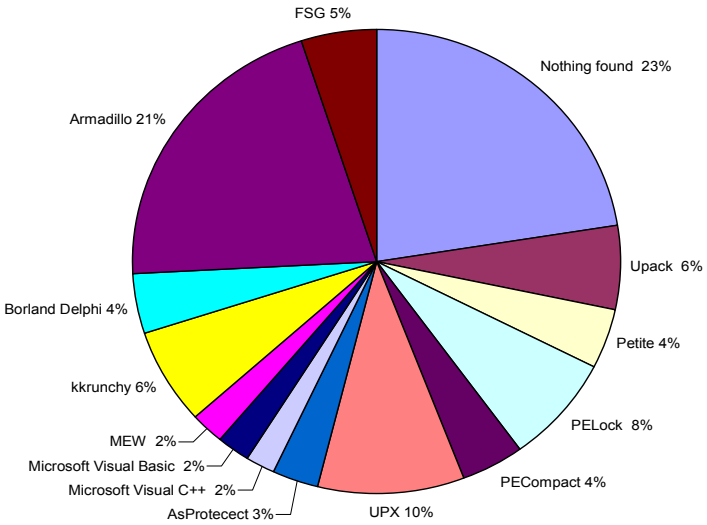

Figure 2: Distribution of obfuscation packers used in malware

creating malware that cannot be detected by AV engines, and their level of sophistication is continuing to grow.

Researchers have been trying to build semi-automated tools for automatically unpacking malware, such as PolyUnpack [23], Renovo [24], OmniUnpack [25], Ether [26], and Eureka [9]. PolyUnpack is an automated unpacking technique for extracting the hidden code through process execution and uses the Windows debugging API to single-step. The second tool, Renovo, is implemented using the QEMU emulator and supports multiple layers of unpacking. However, OmniUnpack uses a coarse-grained execution tracking approach at the page-level protection mechanism available in hardware in order to identify when the code gets executed from a page that was newly modified. Eureka, is similar to OmniUnpack except that Eureka tracks execution at the system call level. Eureka follows statistical bigram analysis and coarse-grained execution tracing method and provides several Windows API resolution techniques that identify API calls based on their functionality in the unpacked code. Lastly, Ether, is based on an application of hardware virtualization extension such as Intel VT, and resides outside the operating system. By studying these semi-automated tools, we observe that none of them are completely meeting the purpose of analyzing the behavior of malware by extracting API call features.

In our experiment conducted on 386 samples of malware, we used PEiD [27], a detector used for most common packers, cryptors, compilers and even signature-based packer detection in PE files. The result is that we found about 77% of these malware to be packed and 23% to be unpacked, as shown in Figure 2. From the result in Figure 2, we can determine that the majority of malware change their byte sequence or 'Signature' by applying packing techniques so as to evade detection by virus scanners.

Step 2: Disassemble the binary executable

All data set files collected were preprocessed for anomaly testing. In order to translate a program into an equivalent high-level-language program based on the binary content, we have used the most reliable disassembly tool used for static analysis, namely, Interactive Disassembler Pro (IDA Pro) [29] since it can disassemble all types of non executable and executable files (such as ELF, EXE, PE, etc.). Also, we have selected the IDA Pro as a component of the automation process of this research work because it automatically recognizes API calls for various compilers and can be further extended with our Python programs and compiled plugins, resulting in incredibly powerful implementation with flexible levels of analysis and control. IDA Pro loads the selected file into memory to analyse the relevant program portion to create an IDA database whose components are stored in four files: .id0 that contains the content of a B-tree-style database, .id1 that contains flags describing each program byte, .nam that contains index information related to program locations, and .til that is used to store information concerning local type definitions to a given database. IDA Pro generates the IDA database files into a single IDB file (.idb) by disassembling and analysing the binary of the file. Our fully-automated system using Python programming language generates .idb automatically from the set of malware samples (Figure 1).

Step 3: Extract API calls

IDA Pro provides access to its internal resources via an API that allows users to create plug-ins to be executed by IDAPro We have made use of SQLite [30], a software library that implements a self-contained transactional SQL database engine. Our Python program automatically runs and creates the plugin to use SQLite with IDA Pro for generating the

TABLE 1
MAIN MALICIOUS BEHAVIOUR GROUPS OF API CALL FEATURES

Behaviour	Malware Category	API Function Calls
Behaviour 1	Search Files to Infect	FindClose, FindFirstFile, FindFirstFileEx, FindFirstFileName, TransactedW, FindFirstFileNameW, FindFirstFileTransacted, FindFirstStream, TransactedW, FindFirstStreamW, FindNextFile, FindNextFileNameW, FindNextStreamW, SearchPath.
Behaviour 2	Copy/Delete Files	CloseHandle, CopyFile, CopyFileEx, CopyFileTransacted, CreateFile, CreateFileTransacted, CreateHardLink, CreateHardLink, Transacted, CreateSymbolicLink, CreateSymbolic, LinkTransacted, DeleteFile, DeleteFileTransacted.
Behaviour 3	Get File Information	GetBinaryType, GetCompressed, FileSize, GetCompressedFile, SizeTransacted, GetFileAttributes, GetFileAttributesEx, GetFileAttributes, Transacted, GetFileBandwidth, Reservation, GetFileInformation, ByHandle, GetFileInformation, ByHandleEx, GetFileSize, GetFileSizeEx, GetFileType, GetFinalPathName, ByHandle, GetFullPathName, GetFullPathName, Transacted, GetLongPathName, GetLongPathName, Transacted, GetShortPathName, GetTempFileName, GetTempPath.
Behaviour 4	Move Files	MoveFile, MoveFileEx, MoveFileTransacted, MoveFileWithProgress.
Behaviour 5	Read/Write Files	OpenFile, OpenFileById, ReOpenFile, ReplaceFile, WriteFile, CreateFile, CloseHandle.
Behaviour 6	Change File Attributes	SetFileApisToANSI, SetFileApisToOEM, SetFileAttributes, SetFileAttributesTransacted, SetFileBandwidthReservation, SetFileInformationByHandle, SetFileShortName, SetFileValidData

database (.db). We have developed an interface for accessing the database file (.db) so that the results from the assembly code of the malware stored in the database could be used for better binary analysis.

```
#include <idc.idc>
  static main()
  {
   …
  RunPlugin("ida2sqlite3",1);
  Message("Alldone, exiting\n");
  …
  Exit(0);
  }
```

IDASQLit plugin generates eight tables (Blocks, Functions, Instructions, Names, Maps, Stacks, Segments, TargetBinaries), each of them contains different information about the binary content. Function table contains all the recognizable API system calls and non-recognizable function names and the length (start and the end location of each function). Instructions table contains all the operation code (OP) and their addresses and block addresses. Maps table contains the function address and source of block address and the destination of the function address. Names table contains function addresses, the name of the function and the type of the function. Stacks table contains function address, the stack name, and the start and the end address. Segments table contains information that describes each segment in an executable file, segment name (Code, Data, BSS, _idata, _tls, _rdata, _reloc, and _rsrc) and the segment length. Finally, TargetBinaries contain the file name, path name, MD5, and start and the end of analyses.

Step 4: API call mapping and feature analysis

As a reference for this step, we downloaded the Windows application programming interface (API) from The Microsoft Developer Network (MSDN) [31]. We implemented in Python the required processes to compare and match the API from MSDN and the API calls generated in the database (.db of Step 3) for the malware sample set. In addition, to list all the API calls that are associated with malcode and to analyse the features, we have considered the machine opcodes such as Jump and Call operations as well as the function type (import or function). Sample code snippets are as shown below.

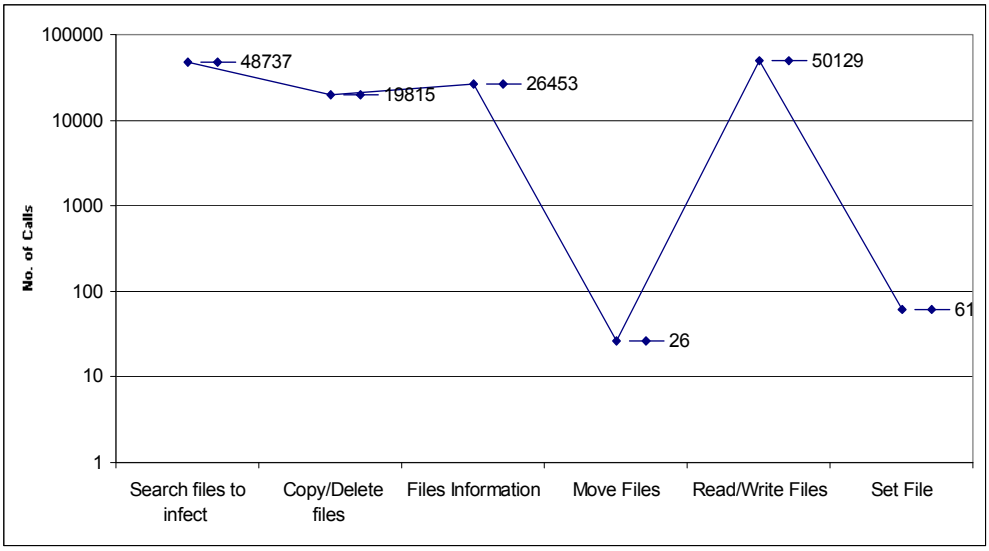

Figure 3: API call distribution of malware samples

```
SELECT function_address,src_block_address,name
from Maps, Names"
Where (op='call' or op='jmp') and (type='import'
or type='function')
```

For the analysis of malware behaviour, we have considered features such as frequency of call, call sequence pattern and actions immediately preceding or after call. Some actions that lead to invalid memory reference or undefined register or invalid jump target help in refining the extracted features for analysis.

We have developed a fully-automated system that integrates well with IDA Pro and SQlite using Python programming to perform all the four steps described above and have performed experimental testing for all the 386 malware samples.

V. Experimental Results

All executable programs, malicious or not, have the goal to perform an action using API calls. Disguised malicious code uses a different, relatively peculiar action which is called suspicious behavior. Behavior identification is becoming a rich area to study and as explained earlier, the malware authors target their malware on the commonly used NTFS by using API functions provided under Win32 environment to implement their functions. A statistical analysis of the Windows API calling sequence reflects the behavior of a particular piece of code. In this research work, the API call features from the binary of a program were extracted and analysed to understand their malicious behavior and finally to classify the program as malicious or benign. The extracted features were subjected to a statistical test to determine the malware class based on suspicious behaviours. As a result of our experimental analysis on the 386 malware samples, we have identified six main groups of commonly used API function call features that are based on the malicious behavior patterns (Table 1) and these are listed below:

1) Search files to infect.
2) Copy/Delete files.
3) Get file information.
4) Move Files.
5) Read /Write files.
6) Change file attributes.

Figure 3 shows the frequency distribution of these six main groups of API calls invoked within the experimental dataset for malicious purposes. From Figure 3, it is clear that the most prominent behavior commonly exhibited by malwares is to infect file through API calls that perform read/write files.

VI. Limitations and Future Work

The system call interface is the facility that the OS offers to user-mode applications. UNIX operating system had a well document clearly defined set of system calls. The MINIX operating system has a system call interface consisting of only 53 routines. Everything that the MINIX operating system is capable of doing ultimately can be resolved into one or more of these system calls. The Window operating system, which refers to its system call interface as the native API of Windows, has not provided an official document for its native API. However, today Windows is the basis of predominant operating systems in use, such as Windows 2000, Windows XP, Windows Server 2003, Windows Server 2008, Windows Vista, Windows 7 [32] [33]. Therefore, this research is a step towards addressing malware that try to target on windows operating system with the view of affecting more computer users. Hence, this research work has limitations in its application exclusively to malware that make use of Windows API calls.

Future work entails extracting binary N-gram features complement the API cal features and to train the classifier resulting in building a model using support vector machine (SVM). We would also test the model against large sets of malware samples for verifying the accuracy of the modelled system.

VII. CONCLUSION

A statistical analysis of the Windows API calling sequence reflects the behavior of a particular piece of code. In this research project, the API calls from the binary of a program were extracted to analyse the most common malware behavior patterns and to classify program executables as malicious or benign. The extracted calls were subjected to a statistical test to determine the malware class based on suspicious behavior. The entire static detection process was a fully-automated system and a four-step methodology was adopted for developing the system. Experimental tests were conducted using 386 samples of malware and we have arrived at six main categories of suspicious behavior of API call features. These being i) Search files to infect, ii) Copy/Delete files,.iii) Get file information, iv) Move Files, v) Read /Write files and vi) Change file attributes. Among these, API calls for Read /Write files were predominantly used by malware as a vehicle to infect the program.

ACKNOWLEDGEMENTS

This research was conducted at the Internet Commerce Security Laboratory and was funded by the State Government of Victoria, IBM, Westpac, the Australian Federal Police and the University of Ballarat. We wish to thank Seokwoo Choi and Robert Layton for their technical help and invaluable comments.

REFERENCES

[1] G. McGraw and G. Morrisett, “Attacking malicious code: A report to the infosec research council”, IEEE Software, 2000, 17(5), 33–44.

[2] Bergeron, J.; Debbabi, M.; Desharnais, J.; Erhioui, M.; Lavoie, Y. & Tawbi, N., "Static detection of malicious code in executable programs", In Symposium on Requirements Engineering for Information Security, Citeseer, 2001, 184-189.

[3] Vinod, P.; Jaipur, R.; Laxmi, V. & Gaur, M., “Survey on Malware Detection Methods”, Hack. 2009, 74.

[4] M. Christodorescu and S. Jha, “Testing malware detectors”, In Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004), Boston, MA, USA, ACM Press, 2004, 34–44.

[5] Christodorescu, M. & Jha, S., “Static analysis of executables to detect malicious patterns”, Proceedings of the 12th conference on USENIX Security Symposium, 2003, 12, 169-186.

[6] Alazab, M.; Venkatraman, S. & Watters, P., “Effective digital forensic analysis of the NTFS disk image”, Ubiquitous Computing and Communication Journal, 2009, 4, 1.

[7] Keizer, G., “Symantec false positive cripples thousands of Chinese PCs” http://www.computerworld.com/s/article/9019958/Symantec_false_positive_cripples_thousands_of_Chinese_PCs?intsrc=hm_list, August 2009.

[8] Alazab, M., Venkatraman, S. and Watters, P., ‘Digital Forensic Techniques for Static Analysis of NTFS Images, Proceedings of International Conference on Information Technology (ICIT2009). IEEE Computer Society, ISBN 9957-8583-0-0. 2009.

[9] Sharif, M.; Yegneswaran, V.; Saidi, H.; Porras, P. & Lee, W., "Eureka: A framework for enabling static malware analysis", Computer Security - ESORICS, Lecture Notes in Computer Science LNCS, Springer, 2008, 5283/2008, 481-500.

[10] Tamada, H.; Okamoto, K.; Nakamura, M.; Monden, A. & Matsumoto, K.,M, "Dynamic software birthmarks to detect the theft of windows applications", International Symposium on Future Software Technology, 2004.

[11] Okamoto, K.; Tamada, H.; Nakamura, M.; Monden, A. & Matsumoto, K., "Dynamic Software Birthmarks Based on API Calls", IEICE Transactions on Information and Systems, 2006, J89-D, 1751-1763.

[12] Choi, S.; Park, H.; Lim, H. & Han, T., "A static birthmark of binary executables based on API call structure", Advances in Computer Science--ASIAN 2007. Computer and Network Security, Lecture Notes in Computer Science LNCS, Springer, 2008, 4846/2008, 2-16.

[13] Wang, C.; Pang, J.; Zhao, R.; Fu, W. & Liu, X., “Malware Detection Based on Suspicious Behavior Identification”, Education Technology and Computer Science, International Workshop on, IEEE Computer Society, 2009, 2, 198-202.

[14] Li, W.; Wang, K.; Stolfo, S. & Herzog, B., “Fileprints: Identifying file types by n-gram analysis”, the Proceedings of the 2005 IEEE Workshop on Information Assurance and Security, 2005.

[15] Venkatraman, S., "Autonomic Context-Dependent Architecture for Malware Detection", Proceedings of International Conference on e-Technology (e-Tech2009), International Business Academics Consortium, ISBN 978-986-83038-3-6, 8-10 January, Singapore, 2009, 2927-2947.

[16] Microsoft, 2007, “Understanding Anti-Malware Technologies”, http://download.microsoft.com/download/0/c/0/0c040c8f-2109-4760-a750-96443fd14ef2/Understanding Malware Research and Response at Microsoft.pdf, August 2009 .

[17] Bruschi, D.; Martignoni, L. & Monga, M., “Detecting self-mutating malware using control-flow graph matching”, Lecture Notes in Computer Science, Springer, 2006, 4064, 129.

[18] MetaPHOR, http://securityresponse.symantec.com/avcenter/venc/data/w32.simile.html, August 2009.

[19] P. Ferrie and P. Sz¨or. Zmist opportunities. Virus Bullettin, 2001.

[20] Perriot, F. & Ferrie, P., “Principles and practise of x-raying”, Virus Bulletin Conference, 2004, 51-56.

[21] Turner, D. Semantic internet security threat report: Trends for january 06 - june 06. X . http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf. 2006.

[22] NEOx, PE Tools, http://www.uinc.ru, March 2010.

[23] Royal, P.; Halpin, M.; Dagon, D.; Edmonds, R. & Lee, W., "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware", IEEE Computer Society, the 22nd Annual Computer Security Applications Conference (ACSAC'06), 2006, 289-300.

[24] Kang, M.; Poosankam, P. & Yin, H. " Renovo: A hidden code extractor for packed executables", Workshop On Rapid Malcode WORM'07 Proceedings of the 2007 ACM workshop on Recurring malcode, 2007, 46 – 53.

[25] Martignoni, L.; Christodorescu, M. & Jha, S., "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware", Proceedings of the Annual Computer Security Applications Conference (ACSAC), 2007.

[26] Dinaburg, A.; Royal, P.; Sharif, M. & Lee, W., "Ether: Malware analysis via hardware virtualization extensions", Proceedings of the 15th ACM conference on Computer and communications security, 2008, 51-62.

[27] Snaker, Qwerton, Jibz & xineohP, PEiD, http://www.peid.info/, 2008.

[28] Yan, W.; Zhang, Z.; Ansari, N. & Micro, T., "Revealing packed malware", IEEE Security & Privacy, 2008, 6, 65-69.

[29] IDA Pro Dissasember, DataRescue, An Advanced Interactive Multi-processor Disassembler, http://www.datarescue.com, October, 2009.
[30] SQLite, www.sqlite.org/, February 2010.
[31] Windows API Functions, MSDN, http://msdn.microsoft.com/en-us/library/aa383749%28VS.85%29.aspx. January 2010.
[32] Alazab, M., "Investigation techniques for static analysis of NTFS file system images", 2009 Annual Research Conference, Internet Security, University of Ballarat.
[33] Purcell, D. & Lang, S., "Forensic Artifacts of Microsoft Windows Vista System", Lecture Notes in Computer Science, Springer, 2008, 5075, 304-319.
[34] Wang, C.; Pang, J.; Zhao, R. & Liu, X. "Using API Sequence and Bayes Algorithm to Detect Suspicious Behavior", 2009 International Conference on Communication Software and Networks, 2009, 544-548.
[35] Choi, S.; Park, H.; Lim, H. & Han, T., "A static API birthmark for Windows binary executables", Journal of Systems and Software, Elsevier, 2009, 82, 862-873.
[36] Stolfo, S.; Wang, K. & Li, W., "Towards Stealthy Malware Detection", Malware Detection, Springer, 2007, 27, 231-249.
[37] Sun, L.; Ebringer, T. & Boztas, S., "An automatic anti-anti-VMware technique applicable for multi-stage packed malware", Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on, 2008, 17-23.