Quest Journals
Journal of Software Engineering and Simulation
Volume 2 ~ Issue 11 (2015) pp: 01-05
ISSN(Online) :2321-3795 ISSN (Print):2321-3809
www.questjournals.org

Research Paper

On the Vulnerability of Behaviour-based Malware Detection Methods

Amir Mohammadzade Lajevardi[1], Saeed Parsa[2], Mohammad Javad Amiri[3]
[1]*PhD Student, Software Engineering Group, Sharif University of Technology, School of Computer Engineering, Tehran,*
[2]*Associate Professor, Software Engineering Group, Iran University of Science and Technology, School of Computer Engineering, Tehran,*
[3]*PhD Student, University of California Santa Barbara, Department of Computer Science, California, USA,*

Received 18 October, 2015; **A**ccepted 05 November, 2015

ABSTRACT:- *Malwares can be detected by their malicious behaviours. Most malware analysis approaches are based on behavioural detection methods. Malware behaviour known by its sequences of API calls. There are three approaches to hook API calls: kernel hooking, user hooking and Filter Drivers. Most of anti-malware tools use kernel hooking for intercepting and logging API calls. In this article, a Shadow System Call approach is discussed which could hide malware's system calls. We use vulnerabilities concerning behaviour based malware detection methods. The problem in API interception is to keep the logs of the sequence of API calls. By taking the advantage of this problem, we have made it possible to disable behavioural analysis of well-known malware detectors -such as Norton, ESET, Bitdefender and Kaspersky- through stopping their API call interceptor and terminating their process by controlling the access to SSDT table and NtTerminate Process API.*

Keywords:- *Behavior, Anti-Malware, Interception, SSDT, Hook*

I. INTRODUCTION

In this article, the design and implementation of a kernel driver to automatically stop behavioural detection of programs is presented. Malware detectors rely on behavioural analysis to detect newly born malicious codes and viruses. Behavioural-based analysis has been introduced to cover the shortcomings of signature-based antivirus solution to recognize obfuscated and new unknown malware specimens. However, there is a major drawback concerning behavioural approaches that are the main contribution of this article. The drawback concerns the reliance of behavioural analysis on system calls, which could be simply obfuscated. Since behavioural detection methods need to monitor API calls, the vulnerabilities of monitoring approaches have damaged on-line behavioural analysis.

In software security area, behaviour is modelled as a sequence of system calls [1]. Therefore, behavioural obfuscation focuses on hiding the sequence of system calls. However, despite the variety of behavioural obfuscation techniques such as Staged API [2] and Push-Calc-Ret [2], malware detectors can simply detect any obfuscated behaviour at run time. Code level behaviour can be detected by simply hooking the entry point of system APIs to detect any sequence of system calls made by a process at run time. The hooking can be made through patching API addresses in the system dispatch tables that are built and controlled by the operating system.

System calls can be detected through static and dynamic analysis of programs. Applying de-obfuscators can detect most obfuscated system calls statically, provided that the obfuscation technique is already known. In general, all obfuscated system calls can be detected at run time when the target system function is patched. Complex malwares pack their System calls, so when a sample cannot be unpacked, memory dumps may be used to provide insight into its behaviour. In contrast, in dynamic analysis, in-depth analysis of packed threats requires the knowledge of the API functions called during execution.

In windows operating system, the security defence software's implement system call interception by replacing system function entry addresses in specific table called System Service Dispatch Table (SSDT) with its own log keeping function addresses [3-4].

*Corresponding Author: Mohammad Javad Amiri

[3]*PhD Student, University of California Santa Barbara, Department of Computer Science, California, USA,*

SSDT restoration method, discussed in [5], restores SSDT from user-space. However, new operating systems prevent user mode programs from accessing kernel space. Another restoration method discussed in [6] restores SSDT through a number of API calls. The difficulty is that API calls can be simply intercepted through SSDT by malware detectors. In this article, to restore SSDT, we use low level system calls that are not intercepted by known malware detectors. After SSDT restoration, we succeed to terminate malware detectors processes.

The rest of this paper is organized as follows: Section 2 presents proposed method for restoring SSDT Table. In the section 3 this method evaluated in two different manners and finally, in section 4 the conclusion is described.

II. DISCUSSION

When a program invokes a system API, the windows operating system seeks the API entry point address in a table called System Service Dispatch Table (SSDT) [4]. While starting up the windows operating system, SSDT is launched into the kernel space of main memory. SSDT plays the role of a mediator between user level API and kernel level API. SSDT contains not only the index table of addresses, but also other useful information such as the base address, the number of functions, and so on. The structure of SSDT includes four members: ServiceTableBase, ServiceCounterTableBase, NumberOfService, and ParamTableBase. ServiceTableBase points to the base address of system service descriptor table. Service-CounterTableBase points to another index table that contains the call number of each service called. NumberOfService describes the number of service functions that the current operating system supports. ParamTableBase points to the table that contains the number of bytes of parameters for each service [4, 7].

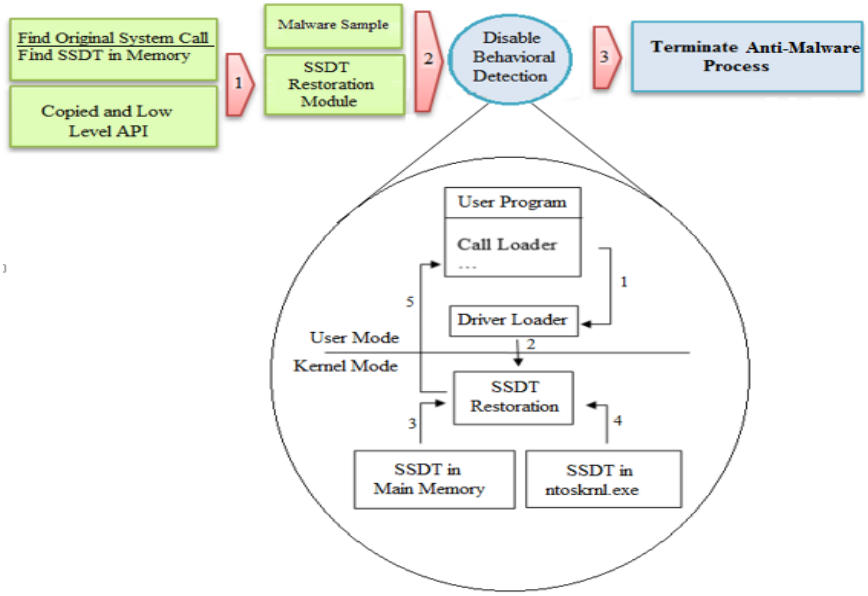

Fig. 1. Suggested Method Diagram

Suggested method diagram has shown in fig. 1. In order to stop behaviour based analysis of programs; the following procedure described in the three main steps could be applied:

1) Find the original address of API entry points: To obtain the base memory address of APIs, the executable code ntoskrnl.exe should be loaded in memory. We should parse ntoskrnl.exe and create a section for it in memory. We can use ZwMapViewOfSection and ZwCreateFileMapping functions for file mapping and paring ntoskrnl in memory. But these functions are in SSDT and may be hooked by anti-malware tools, so we should use low-level API functions like Mm-family API for file mapping. These functions are exported by ntoskrnl, and can easily be accessed using C language. After parsing ntoskrnl sections we found Original SSDT APIs.

2) Find kernel SSDT in memory: in this stage, the address of SSDT, used by the operating system for its system calls, is found in the main memory. The SSDT table is addressed in a C struct called SDT. SDT can be accessed through a global variable named KeServiceDescriptorTable. This variable is exported by ntoskrnl.exe file, located in System Drive\Windows\System32 path.

Ke Service Descriptor Table includes the relative address of SDT. We use "__declspec(dllimport) SDT Ke Service Descriptor Table" instruction for finding the address of SDT. SDT is described as below:

```
typedef struct ServiceDescriptorEntry
{
        PDWORD ServiceTableBase;
        PDWORD ServiceCounterTableBase ;
        DWORD NumberOfService ;
        PBYTE ParamTableBase;
} SDT;
```

3) Restore all hooked API addresses to their original address: After the hooked SSDT is found, its API addresses are restored to their original value, found in stage 1. In order to restore addresses, we used "memcpy" function call in C.

The above-mentioned procedure could be applied to stop behaviour analysis of programs. As shown in fig. 2., we have developed a device driver to hide native system calls. The driver is activated by a program and could be executed through any application. The driver disables the behavioural engine of anti-malware tools. In this situation, malwares could call any sequence of APIs without any restriction.

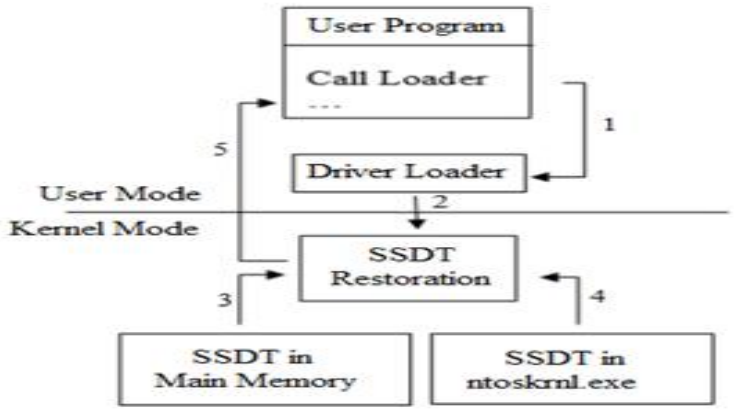

Fig. 2. Add Represented Module to Malware Sample

Most anti-malwares protect themselves by hooking NtTerminateProcess API. In this way, if any malware attempts to terminate the anti-malware process it will be detected. After unhooking NtTerminateProcess API, we could simply terminate anti-malware processes. But some anti-malware tools like Kaspersky and Bitdefender use specific device drivers for their self-protection. In these cases, after disabling the behavioural detection of malwares, we could simply unload the self-detector driver and then terminate anti-malware processes. Some other anti-malwares such as Bitdefender hook SSDT in specific periods of time, so we found this period and our driver restores SSDT periodically after Bitdefender hooks it. Other malware detector tools such as ESET are controlled by a distinct hidden supervisory process that monitors their execution and restarts their processes immediately after the malware detector is terminated. To neutralize the supervisory process, we hook the NtCreateProcess API to ignore the supervisory process requests for activating the ESET main process.

III. EVALUATION

The In this section, in order to evaluate our proposed method, our experiments with neutralizing and terminating seven well-known anti-malwares are presented. Also the pros and cons of our proposed obfuscation technique in comparison with some other known ones are presented.

To evaluate the efficiency of our device driver, we run it under Windows 7 and Windows 8 (32bit) operating system environments. The driver firstly scanned the system service descriptor table to detect hooked addresses. There were 284 NT-family system calls addressed from within SSDT. The device driver detected all the API Addresses altered and hooked by API monitors and anti-malwares tools. In table 1 the number of hooked system calls by seven well-known anti-malware tools is shown.

Table 1. Number of Hooked API Addresses in Anti-Malware Tools

Antivirus and malware Detector tools	Number of Hooked API Addresses
Avast7_2013	**20**
Avira.Internet.Security.2013.v13	**24**
Bitdefender.Total.Security.2013.v16	**37**
Kaspersky.Internet.Security.2013.v13.0	**60**
Norton.Antivirus.2012.v19	**35**
ESET.Smart.Security.5.2.15	**19**
Panda.Internet.Security.2012.v17	**1**

All these anti-malwares hook NtTerminateProcess API to prevent malwares from terminating their process. The reason is that if a malware attempts to terminate their process they could quickly detect and ignore the system call. After unhooking the NtTerminateProcess API, we could terminate all the seven anti-malware processes named in table 1. To test the capability of our device driver in unhooking the SSDT, we collected 22 well-known malwares from [8]. Executing the malwares we found none of the seven anti-malwares could track and intercept the system calls made by these 22 samples.

Table 2. Compare Represented Method with Related Methods

Obfuscation Method	Advantage	Disadvantage
Copied and Substituted API [2]	Independent from the OS APIs	**Is detected by pattern recognition techniques. Code size will increase.**
Restore SSDT by Native API calls [6]	Restore SSDT	**Can simply detected by SSDT hooking**
Restore SSDT by user mode program [5]	Independent of the Loading Driver	**Ineffective against new operating systems that prevent user mode program to access to kernel space**
Proposed Method	Independent of the SSDT APIs, few changes in malware codes	**Need to load device driver at malware run time**

As shown in table 2 our proposed method doesn't use any SSDT API's for restoration and need few changes in malware source code.

IV. C CONCLUSION

In this article, a method to hide system calls through low-level APIs and SSDT restoration is presented. There are many system call obfuscation techniques that hide system calls, but these techniques just stop API detection through static analysis approaches. Also, these techniques lead to increase in size of code. But the represented module just adds 70kB to the malware sample and this added code is independent of malware size. If anti-malware tools want to prevent SSDT restoration, they should hook low-level system calls like Mm-family APIs, because most malwares use these functions to achieve their malicious goals.

REFERENCES

[1]. Idika, N., & Mathur, A. P., *A Survey of Malware Detection Techniques, Department of Computer Science,* Purdue University, West Lafayette, 2007.

[2]. *A Museum of API Obfuscation on Win32,* Symantec Corp, 2011, http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/a_museum_of_api_obfuscation_on_win32.pdf. Latest Access Time for the website is 15 October 2015.

[3]. Egele, M., Scholte, T., Kirda, E., & Kruegel, C., *A survey on automated dynamic malware-analysis techniques and tools*, vol. 44, pp.6-8, In Proceedings of ACM Computer, 2012.

[4]. Schreiber, S. B, *Undocumented Windows 2000 Secrets, A Programmer's Cookbook*, Addison Wesley Longman Publishing Co, Boston, MA, USA, 2001.
[5]. Zhang, J., Liu, S., Peng, J., & Guan, A., *Techniques of user-mode detecting System Service Descriptor Table*, In Proceedings of the 2009 13th International Conference on Computer Supported Cooperative Work in Design, IEEE Computer Society, pp. 96-10, Washington, DC, USA, 2009.
[6]. Zhang, Y., & Bi, H., *Anti-rootkit Technology of Kernel Integrity Detection and Restoration*, Network Computing and Information Security (NCIS), vol. 1 pp. 276- 278, 2011.
[7]. *Hooking the native API and controlling process creation on a system-wide basis*, http://www.codeproject.com/Articles/11985/Hooking-the-native-API-and-controlling-process-cre, Latest Access Time for the website is 17 September 2015.
[8]. http://www.virussign.com/ Latest Access Time for the website is 10 October 2015.