2015 IEEE 2015 International Conference on Computer, Communication, and Control Technology (I4CT 2015), April 21 - 23 in Imperial Kuching Hotel, Kuching, Sarawak, Malaysia

In Memory Detection of Windows API Call Hooking Technique

Syed Zainudeen Mohd Shaid
Faculty of Computing,
Universiti Teknologi Malaysia
81310 UTM Johor Bahru, Johor, Malaysia
szainudeen@utm.my

Mohd Aizaini Maarof
Faculty of Computing,
Universiti Teknologi Malaysia
81310 UTM Johor Bahru, Johor, Malaysia
aizaini@utm.my

Abstract— **API call hooking is a technique that malware researchers use to mine malware's API calls. These API calls is used to represent malware's behavior, for use in malware analysis, classification or detection of samples. In this paper, analysis of current Windows API call hooking techniques is presented where surprisingly, it was found that detection of each technique can be done trivially in memory. This could lead to malware being able to sense the presence of API call hooking techniques and modifying their behavior during runtime. Suggestions for a better API call hooking technique are presented towards the end of the paper.**

Keywords-component; Malware; API cal; API hooking

I. INTRODUCTION

Application Programming Interface (API) calls is a popular dynamic feature used in malware research and this can be clearly seen in the number of research papers that chose API calls as a dynamic feature for representing malware's behavior [1][2][3][4][5]. API calls are basically functions that a program (malware included) invokes during its execution. APIs simplify the work of writing portable code by abstracting lower level functions into a set of stable, portable interface that programmers can use to ensure reliable operation of their software. The idea of using API calls as a dynamic feature of malware depends on the fact that different programs might work in a different way and thus, use different sets of APIs. The difference could be in the APIs that was called, the parameters that was used in an API call (if any), or even in the order in which the APIs are invoked.

There are several types of API as illustrated in Figure 1. Basically, there are language specific APIs, and Operating System (OS) specific APIs. The OS specific APIs, in case of Windows OS, can be further divided into User-mode APIs, Native APIs, and Kernel-mode APIs. Table 1 shows sample API calls at each level that performs the file open operation. Language specific APIs are OS dependent and the APIs are implemented by calling the underlying OS's API. Linux based systems have libc or GNU C library (glibc) that will expose the C APIs while in Windows, the C Runtime (CRT) library is exposing the C APIs. These libraries call the underlying APIs in order to implement a specific function. This can be verified by tracing an API call in runtime using a debugger. For example, the fopen API call in Windows, will at some point, result in a call to the *CreateFile* API, which will later call the native API *NtCreateFile* before a switch to kernel mode code is being made.

API names are very descriptive and the name itself describes the function that it carries. Capturing API calls made by malware during execution will shed some light on its activities which can be used as an indicator to represent its behavior. The issue is, can we trust the list of APIs obtained from current API call hooking techniques?

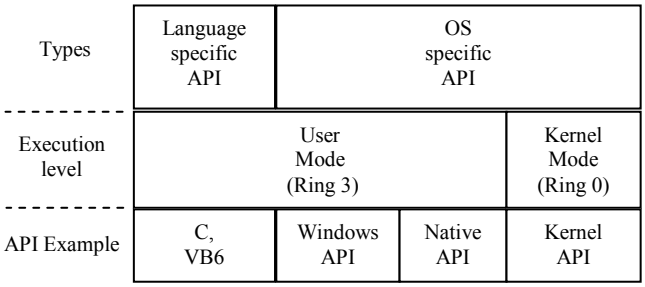

Figure 1. APIs in Windows OS

TABLE I. SAMPLE APIS

API Example	***Sample API call***
C	*fopen*
VB6	*Open* filename *For Input As* #fileNum
Windows API	*CreateFile*
Native API	*NtCreateFile*

II. MOTIVATION

Malware can detect the environment in which they are being executed and determine whether they are being analyzed or not. Such malware are known as 'environment-aware' malware and can modify their behavior during runtime in order to evade detection. There have been research done on malware detecting the presence of Virtual Machine (VM) and Emulators

978-1-4799-7952-3/15/$31.00 ©2015 IEEE

[6][7]. VM and emulators are among the technologies used to provide a safe execution environment for executing malware samples. API call monitoring on the other hand, is a technique used to gather a list of API calls issued by malware during runtime [8][9][10]. The list can be used to give an insight into the behavioral aspects of a malware sample. If malware can detect the presence of API hooking techniques, there is a possibility that malware can change its behavior. Therefore, this research focuses on the possibility of detecting the presence of API call hooking techniques in memory, during runtime.

III. Existing Windows API Call Hooking Technique

API call hooking is possible in both Windows and Linux. However, this paper will focus on API call hooking for the Windows OS, specifically on user mode API calls. There are several well-known techniques for user mode API call hooking. These are Import Address Table (IAT) Hook [8], Debugger Hook [9], and Inline Hook [10]. The following subsections discuss each of the abovementioned techniques.

A. Import Address Table Hook

Import Address Table (IAT) Hook is an API hooking technique that works by modifying the IAT [8], located at the header of the Portable Executable (PE) file [11]. PE is the file format used by Windows executable files [11]. Inside a PE file header, there contains a data structure called the Import Address Table (IAT) which is used by Windows to link application with APIs [11]. These APIs are actually exported functions of Dynamic Link Libraries (DLL). The main idea of IAT hooking is to modify these pointers so that instead of pointing to APIs, it will point to a stub that will log API calls and later forwards them to the actual API [8]. Figure 2 shows a structural view of the PE headers of an executable file, pointing towards the IAT.

An application invokes an API by using the CALL instruction [12]. Figure 3 presents a disassembly of an application (in memory), showing how an indirect call instruction refers to an address inside the IAT. The call instructions in Figure 3 (CALL DWORD PTR DS:[00408000], and CALL DWORD PTR DS:[00408008]) are indirect calls, which will cause code execution to jump to the addresses pointed to by the operand (memory address) of the indirect call instruction.

Therefore, it is possible to hook API calls made using the indirect call instruction by modifying the memory addresses inside the IAT, pointing it to a to a stub that will record the API call being made and redirect code execution back to the original address contained in the original IAT (to maintain the original API functionality) [8].

B. Debugger Hook

Debugger hook works by running a debugger that 'debugs' a target application, waiting for an injected breakpoint to trigger an exception [9]. API hooking is achieved by having the debugger placing a breakpoint at the entry point of an API [9]. Technically, a breakpoint is placed by overwriting the entry point with certain CPU specific instruction (e.g. INT 3 instruction for IA-32) that causes the CPU to throw a Debug Exception whenever the current Instruction Pointer (IP) points to the instruction [12]. The exception will be caught by the debugger and the address of which the exception occurs would be used to identify which API is currently being accessed by the target application.

C. Inline Hook

Inline hook is a technique made famous by Hunt and Brubacher [10]. Inline hook works by modifying the entry point of an API with code that will redirect code execution to a function known as the Detour Function. Detour Function is the first block of code that gets executed once an API has been intercepted. In case of API hooking, Detour Function usually contains code that will log API calls. Overwriting the entry point of the API destroys its original functionality since portions of the original code is lost. Because of this, prior to the modification (to the entry point), instructions from the entry point is first copied to a different memory location. These instructions are referred to as the Trampoline Function. Figure 4 shows the execution flow of a program calling an API, before and after Inline Hook is set to capture API calls.

Once an Inline hooked API is invoked, the entry point of the API will redirect code execution to the Detour Function. The Detour Function will execute and once done, it will jump to the Trampoline Function, executing the original (overwritten) instructions before a jump to the remainder of the

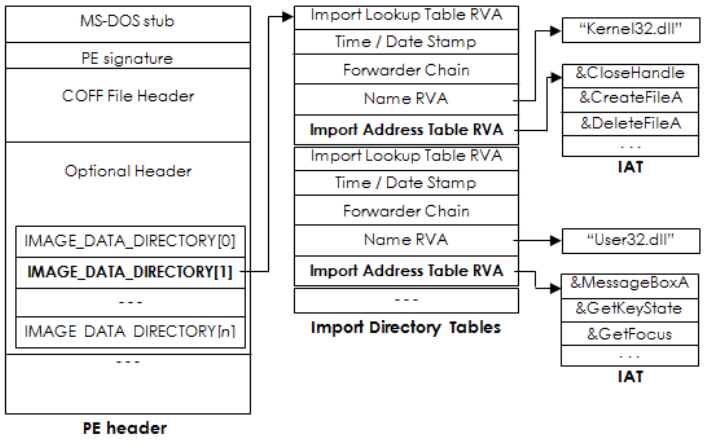

Figure 2. IAT inside a PE file

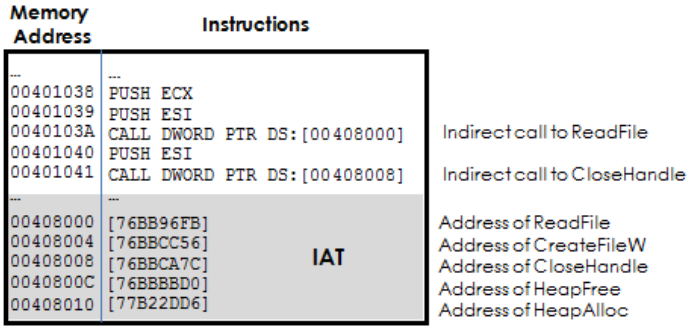

Figure 3. The relationship between indirect calls and IAT

API instruction is being made (see Figure 4). This flow of code execution will ensure that modifications made to the entry point of an API will not affect the original functionality of the API. The above mentioned technique serves as the most basic form of an Inline hook. Actually, a number of variations can be made to the above technique, especially in the type of instructions used to overwrite the entry point with. Below is a list of several possible instructions that can be used at the entry point of an API (on an IA-32 system):

1. Direct JMP instruction.
2. Indirect JMP instruction.
3. PUSH, RET combination.
4. JMP SHORT, JMP NEAR combination

The most interesting variant is the "JMP SHORT, JMP NEAR combination", which involves the use of two JMP instructions. In certain cases, the use of this variant will not require the use of a Trampoline Function. However, the most important rule for this (Inline hooking without a Trampoline Function) to work is the availability of a two byte, 'do-nothing' instruction at the entry point of an API, and 5 unused bytes close to the entry point or around -125 to +129 bytes from the entry point of the API [12]. In Windows, these requirements can be met by compiling an API with the "/hotpatch" option [13] and most of the Windows API are compiled with the "/hotpatch" option (can be observed by disassembling Windows API in memory). The "/hotpatch" option generates 5 padding bytes (using NOP or INT 3 instructions) prior to the entry point of an API, and a two byte 'do-nothing' instruction at the entry point of an API (see Figure 5(a)). Among the popular two byte 'do-nothing' instructions at the entry point of an API (based on an observation in Windows 7 32-bit) are:

1. MOV EDI, EDI
2. MOV EAX, EAX

These two bytes 'do-nothing' instructions are safe to overwrite with a JMP SHORT instruction, pointing to the start of the 5 padding bytes. The 5 padding bytes will be overwritten with a JMP NEAR instruction, pointing to the Detour function.

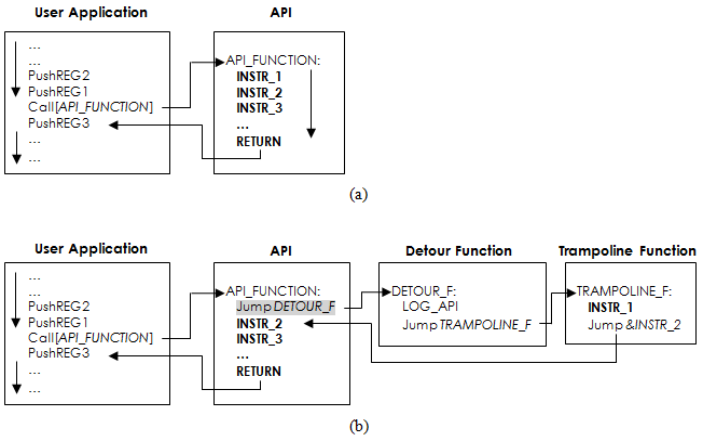

Figure 4. (a) Normal API call code execution flow, and (b) Inline hook in action

This in effect will cause code execution to jump from the entry point of an API to the Detour Function (see Figure 5). The JMP NEAR instruction is needed because the JMP SHORT instruction could not be used to redirect code execution to the Detour Function due to its small jump range. It would be very difficult to find free memory space to place the Detour Function inside the module that contains the API (since most of the memory space would be fully occupied with executable code that implements various APIs) and thus, a JMP NEAR instruction is needed to allow the Detour Function to be placed outside of the API module (this is possible because the range of a JMP NEAR instruction if farther than a JMP SHORT instruction) [11]

IV. TECHNICAL STUDY OF EXISTING TECHNIQUES

A detailed observation on the working of existing API hooking techniques was made. For each API hooking technique, a sample 'Hello World' program is executed and the main memory of the sample program is observed, before and after an API hook has been initiated on the target program.

Among the things that are observed includes changes to the Portable Executable (PE) file header in main memory, changes to the execution environment of a sample, and changes made to the memory region where the APIs being hooked are located.

A. Findings on IAT Hook

Due to the nature of IAT hook which does not modify memory address containing the API functions, there were no changes made to the memory content before and after IAT hook is in place, including to the entry points of APIs. However, there are noticeable changes being made to the IAT of the target application in main memory.

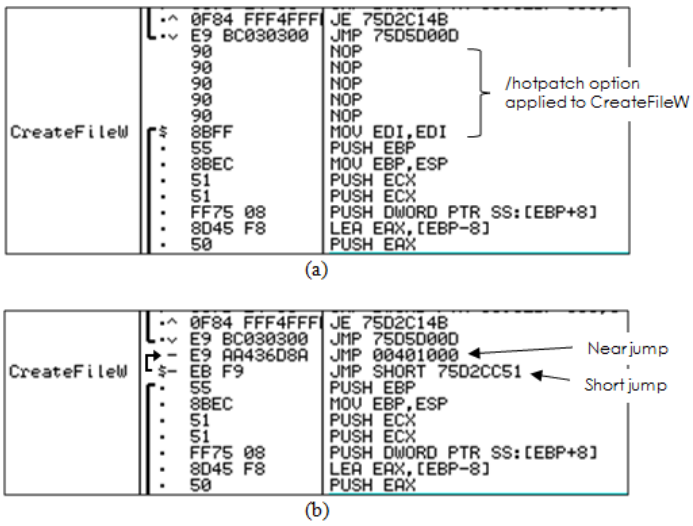

Figure 5. (a) CreateFileW with a 2 bytes 'do-nothing' instruction (MOV EDI, EDI) and 5 NOP instructions as padding (b) CreateFileW has been patched with a short jump (JMP SHORT 75D2CC51) and a near jump instruction (JMP 00401000) that points to a Detour Function (at address 00401000)

Figure 6 shows the change that happens to the IAT of the target application hooked using the IAT hooking technique. It can be seen that *MessageBoxA* is located somewhere in the range on 76F8XXXX while the modified IAT is pointing to memory address in the range of 005AXXXX. The change is very noticeable because the memory address in the IAT (after IAT is in place) is pointing towards a memory address which is outside of the range of address that belongs to the module that contains the *MessageBoxA* API.

B. Findings on Debugger Hook

Due to the use of breakpoints, debugger hook changes the entry point of APIs with a predictable instruction. Figure 7 shows the presence of the INT3 instruction in both of the entry points of *MessageBoxA* and *MessageBoxW*. It was expected that all API hooking techniques causes some change to the memory of a target application. However, the changes made by Debugger Hook are very predictable in that it uses the same instruction (INT3) for patching the entry point of APIs for code redirection purposes. The INT3 instruction is a software break point that causes an exception to be thrown [11]. The instruction is not normally found or used in a normal application since an unhandled exception could cause a program to crash [11]. The presence of a predictable INT3 instruction at the entry point of APIs can therefore signifies the presence of a Debugger Hook.

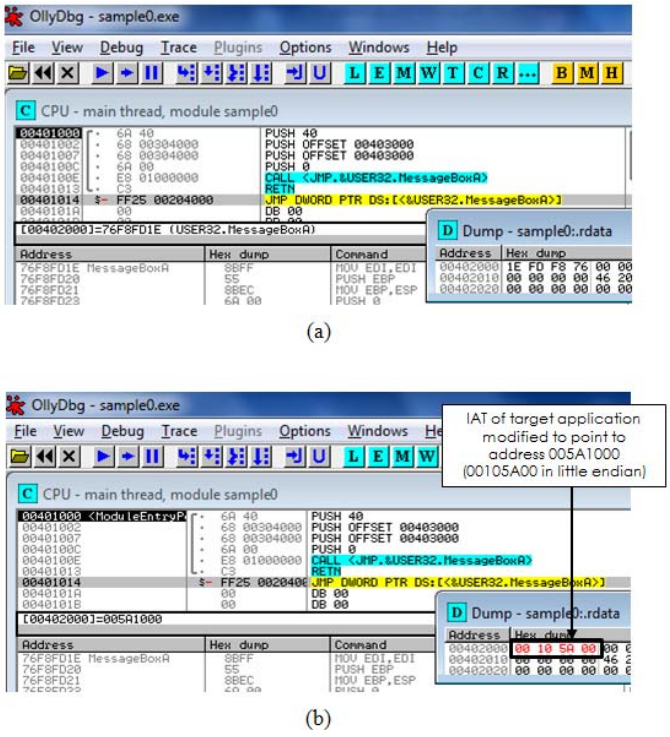

Figure 6. IAT of target application located at address 0040200 (a) before IAT hooking, and (b) the change that happens after IAT hooking

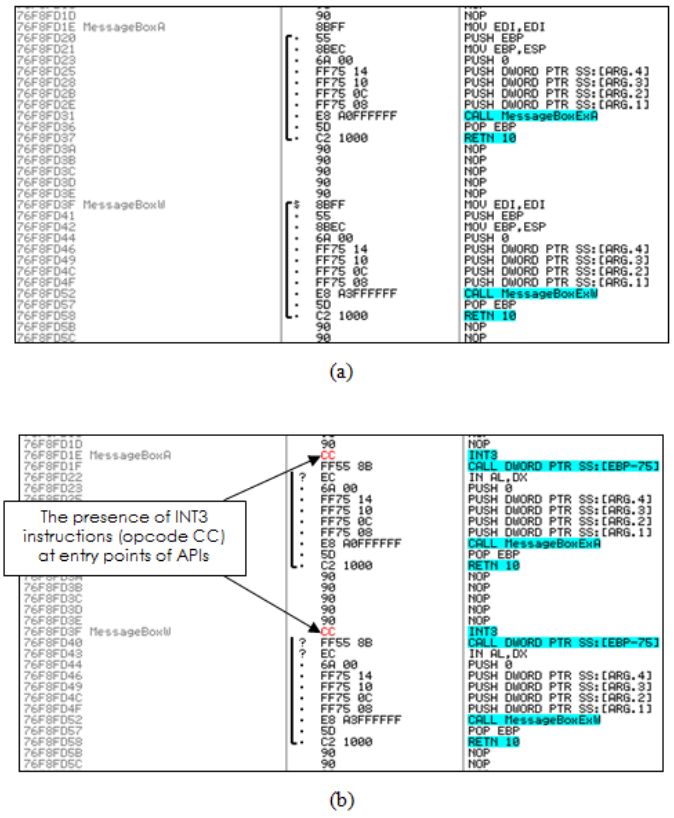

Figure 7. The presence INT3 instruction (opcode CC) at entry points of APIs (b) compared to unhooked APIs in (a)

Another interesting observation made was in the environment of the target application. When a debugger is used to debug a target program, the target program is loaded as a child process inside the debugger. However, when the target program is loaded without the use of a debugger, the target program runs as a child process of 'explorer.exe'. Malware can check for this programmatically during runtime. Besides the abovementioned observations, debugger hook can also be detected using debugger detection techniques [14]. While the presence of a debugger might not signify that an API hooking is in place, malware authors most certainly do not want their code to be inspected and thus, might program their malware to change its behavior in the presence of a debugger.

C. Findings on Inline Hook

As expected, the need to change the entry point of APIs for API call hooking made Inline Hook easy to detect. However, instead of using the INT3 instruction like the Debugger Hook, a jump instruction (JMP) or a PUST-RET instruction combo is normally used (see Figure 8). It is therefore trivial to detect the presence of Inline hook that is, by searching the entry points of APIs for repeated patterns of a jump instruction (JMP) or a PUST-RET instruction combo, or simply, via the absence of a do-nothing instruction at the entry point of APIs.

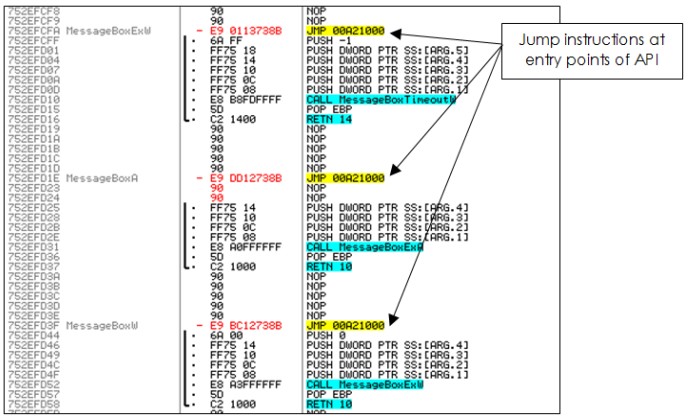

Figure 8. A predictable pattern of jump instructions at entry points of APIs hooked using the Inline Hook technique

V. ANALYSIS OF EXISTING TECHNIQUES

Currently, all existing Windows API call hooking technique can be detected in memory. IAT hooking is easy to detect since changes are guaranteed to be made to the PE's IAT. Besides being easy to detect, IAT hook has a major drawback of not being able to capture dynamically loaded API [15]. Therefore, it is possible for malware to bypass an IAT hook altogether through dynamic invocation of API calls. Debugger hook does not suffer the same limitation but the changes that it made to the execution environment made it even easier to detect. The use of a predictable instruction at the entry point of API means that malware could do a simple if-else comparison to detect the present of breakpoints. Inline hook can be detected using more or less similar method but it seems to have the potential to be upgraded into a better API hooking technique due to the fact that code redirection can be achieved through several different ways. Instead of using predictable jump instruction, probably an obfuscated code for code redirection would yield better stealth. However, care must be taken so as not to overwrite the no-operation instruction placed at the entry point of APIs as that is a trivial way to blow the cover. Obfuscation of code redirection instructions however, might not be feasible on very short pure user-mode API such as the *IsDebuggerPresent* API because of the short length of the API and the fact that such short and simple API can be compared byte-to-byte against a hard-coded version of the API in the malware's binary, as a simple way of detecting possible tampering made to the API, which could imply the presence of an API call hooking techniques in memory.

VI. CONCLUSION

There is a need for a stealthy Windows API hooking technique for capturing malware's API call. The technique must not make predictable changes to specific part of the memory or modify the execution environment of the target application. The technique must be able to redirect code execution from an API using unconventional ways (e.g. using code obfuscation) and do it in a manner that it would difficult for a trivial detection (e.g. if-else comparison). If modification must be made to the memory regions of the API, care must be taken so that the original functionality of the API is preserved.

REFERENCES

[1] Alazab, M., Profiling and classifying the behavior of malicious codes, Journal of Systems and Software, Vol 100, pp 91-102, 2015.

[2] Qiao, Y., et al., CBM: Free, Automatic Malware Analysis Framework Using API Call Sequences, Knowledge Engineering and Management. Springer Berlin Heidelberg, pp. 225-236. 2014.

[3] Elhadi, A. A. E., Maarof, M. A., & Barry, B. I., Improving the detection of malware behaviour using simplified data dependent API call graph. International Journal of Security and Its Applications, Vol.7, pp.29-42, 2013.

[4] Elhadi, A. A., Maarof, M. A., & Osman, A. H., Malware detection based on hybrid signature behaviour application programming interface call graph. American Journal of Applied Sciences, 9(3), pp. 283-288, 2012.

[5] Nataraj, L., Yegneswaran, V., Porras, P., and Zhang, J., A comparative assessment of malware classification using binary texture analysis and dynamic analysis. Proceedings of the 4th ACM workshop on Security and artificial intelligence (AISec '11). ACM, New York, NY, USA, pp. 21-30, 2011.

[6] Ferrie, P., Attacks on Virtual Machine Emulators. Technical report, Symantec Advanced Threat Research. 2006.

[7] Ferrie, P., Attacks on More Virtual Machine Emulators. Technical report, Symantec Advanced Threat Research. 2007

[8] Pietrek, M., Windows 95 System Programming Secrets. CA, John Wiley & Sons Inc. 1995.

[9] Kaplan, Y., API Spying Techniques for Windows 9x, NT and 2000. Retrieved February 27, 2015, from http://www.internals.com/articles/apispy/apispy.htm

[10] Hunt, G., & Brubacher, D., Detours: Binary Interception of Win32 Functions. Proceedings of the Third USENIX Windows NT Symposium. pp. 135-143, 1999.

[11] Microsoft, Microsoft PE and COFF Specification. Technical Report, Microsoft, 2010.

[12] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 2A and 2B: Instruction Set Reference, A-Z. Technical Report. Intel Corp, 2011.

[13] Microsoft, /hotpatch (Create Hotpatchable Image). MSDN. Retrieved February 27, 2015, from http://msdn.microsoft.com/en-us/library/ms173507.aspx,

[14] Branco, R. R., Barbosa, G. N., & Neto, P. D., Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. Black Hat USA, 2012.

[15] Anley, C., et al., G., The Shellcoder's Handbook: Discovering and Exploiting Security Holes. Indiana, Wiley. 2011.