This is the published version:


Available from Deakin Research Online:

http://hdl.handle.net/10536/DRO/DU:30063699

Reproduced with the kind permission of the copyright owner

Copyright: 2008, IEEE
An Evaluation of API Calls Hooking Performance

Mohd Fadzli Marhusin¹, Henry Larkin¹, Chris Lokan¹, David Cornforth¹,²

¹University of New South Wales at Australian Defence Force Academy, ACT 2600, Australia
{m.marhusin, h.larkin, c.lokan}@adfa.edu.au
²CSIRO Energy Technology, Mayfield West NSW 2304, Australia
david.cornforth@csiro.au

Abstract

An open research question in malware detection is how to accurately and reliably distinguish a malware program from a benign one, running on the same machine. In contrast to code signatures, which are commonly used in commercial protection software, signatures derived from system calls have the potential to form the basis of a much more flexible defense mechanism. However, the performance degradation caused by monitoring systems calls could adversely impact the machine. In this paper we report our experimental experience in implementing API hooking to capture sequences of API calls. The loading time of ten common programs was benchmarked with three different settings: plain, computer with antivirus and computer with API hook. Results suggest that the performance of this technique is sufficient to provide a viable approach to distinguishing between benign and malware code execution.

1. Introduction

System call tracing or Applications Programming Interface (API) hooking is a technique that looks for the use of APIs, trapping calls invoked by each program. In intrusion detection and malware detection research, system calls has been used as an option for the source of data analysis. It is believed that malware will generate sequences of API calls that differ from benign programs. Also, it is assumed that for a set of malware which shares common malicious behavior and method of propagation, there is a concise set of sequences of API calls generated by malware which could be used to detect them all. The accuracy of the detection mechanism is subject to the effectiveness of the detection algorithm.

In this paper we investigate the speed penalty imposed on a computer which has API hooking mechanism as compared to a computer without it. This is part of our larger research[1] towards improving the effectiveness of the malware detection technique using sequences of API calls.

2. Problem Statement

Intrusion detection research has employed analysis based on various sources such as audit trails, network traffic and kernel API calls [2]. In malware detection research, a number of solutions has been attempted including checksum, heuristic, integrity shell, string checker, system call tracing, machine emulators, logic analyzers, network sniffer, software certification etc [3],[4]. Each solution has its strengths and weaknesses.

As reported in the Microsoft Developer Network site [5], hooking operating system’s (OS) APIs would hamper the performance of the OS. Thus, it has been suggested to minimize the number of hooks to achieve the hooking goal while preserving a reasonable computer performance. Ritcher[6] suggested that there are seven ways to inject a Dynamic Linked Library (DLL) as a means of monitoring systems calls. One of the techniques implemented in [7] which replaced an Internet Explorer’s real DLL with a bogus one. The fake DLL relays messages sent from and to the original DLL. Doing it that way, they can analyze and log the message pattern of the safe system calls. Forrest et al. [8] fragmented long API calls into several shorter API calls signatures. They looked for similar patterns in the shorter sequences of system calls in code execution. Yangfang et al. [9] recorded all API calls attempts and generated API calls signature. The signature also contains detailed information such as total number of called functions made by that process. They estimated a level of confidence to indicate whether a particular sequence falls into the category of malicious or not, employing a detection process called association rule based classification. Other work presented in [4] monitored several potential APIs used by malware.
such as registry, file system, scripting host, system and communication APIs. Changes made by programs are tracked and recorded so that it can be undone should malicious API calls be detected later.

The aim of this paper is to report the performance of a computer that has a mechanism that traps API calls invoked by programs. We benchmarked computer performance by running programs with three different settings:

1. plain
2. computer with antivirus
3. computer with API hook program

We ran ten programs at our test machine. We recorded the loading time of the programs without antivirus and API hook programs. We also recorded the loading time of the programs which had antivirus installed. We also recorded the loading time of a program that has a set of API hooks being implemented. Our objective is to see whether API hooks hamper computer performance or not in a given context of a limited number of hooks. We also would like to measure how far the performance differs between a computer that has the latest version of the popular antivirus installed and a computer that only has a set of API hooks. While making that statement, note that we do not hook all functions in the OS’s API. We only select a number of functions of our choice from kernel32.dll as listed in the next section. Even though the antivirus program uses file signature based detection technique, it might implement some API hooks as well but such details are not made public. As such, the outcome of this experiment is desirable for us and others to know.

This paper attempts to answer the following questions:

1. Does a hooked program run slower than a plain program?
2. Does a program execute slower in a computer that has signature and heuristic based detection antivirus than in a computer that does not have one?
3. Does a computer that has a hooking program (without antivirus program installed) run slower than a computer that has an antivirus program?

3. Experiment Planning & Operation

As it was reported earlier that implementing API hooks tended to slow a computer performance, we presumed that hooking causes a computer to perform slower than a computer without one. We tested this with ten programs that are widely used by many users. The programs are:

1. Ms Word 2003
2. Ms Excel 2003
3. Ms Outlook 2003
4. Mozilla Firefox 3.0
5. Internet Explorer 7 (IE)
6. Windows Media Player 11 (M.Player)
7. Winamp 5.54
8. Yahoo Messenger 8.1 (YM)
9. Windows Live Messenger 8.5 (Live)
10. Age Of The Empires II Trial (AOE2)

The environment used for the test consists of a machine with Intel(R) Pentium(R) M processor 1600 MHz, 512 MB RAM and 40 GB hard disk. Windows XP SP2 installed on the machine.

Prior to running every experiment, the physical and Virtual PCs’ hard disk was defragmented as necessary. This step is essential to avoid the hard disk’s fragmented disk surface becoming the actual cause of performance degradation. Enabling the hooking mechanism, we created our hook program [1] written in C#.NET in Visual Studio 2008. We utilized a framework that eased the hooking efforts. The framework could be obtained at [10]. The advantage of using the framework was that we could quickly develop the hooking program. We also used AutoIt v3 [11] to help us automate the execution process.

For all of the programs, we measured the time in seconds starting from the moment the program was executed until its graphical user interface became an active window. Each program executed thirty rounds with several second intervals between each execution. The whole process was then repeated two more times, giving three sets of 30 rounds for each program, in order to assess the consistency of the results. It turned out that results from three repetitions were generally consistent. This is important because if we had to do more repetitions it would be very time consuming. The tests involved many file creations and deletions, which required frequent disk defragmentation before subsequent tests.

We were interested to report the time differences that occurred throughout the 30 rounds for those three settings. In this way we benchmarked the computers performance with three different settings namely plain, computer with antivirus and computer with API hook program that logs the API functions’ name, discarding the numerous parameters returned from each invocation. For each program, we prepared a script to launch the program and close it back when its GUI became activated. The program was hooked as soon as it was executed. The hooking program detected the existence of the target executable by monitoring its memory space created in the memory.

At this stage, we only opted to hook kernel32.dll. Kernel32.dll contains the OS’s core functions for many basic operations. Our interests were only the functions
related to create, open, move, delete and replace file operations. We were interested to monitor the socket based API which allows network communications. We were also interested to capture registry operations, since a large number of malware programs hook into the registry. These operations are provided by advapi32.dll, but the last two were not included in this experiment for simplicity. The following is the list of functions of interest:

Kernel32.dll
1. CopyFileA
2. CopyFileExA
3. CopyFileExW
4. CopyFileW
5. CreateFileA
6. CreateFileW
7. DeleteFileA
8. DeleteFileW
9. MoveFileA
10. MoveFileExA
11. MoveFileExW
12. MoveFileW
13. MoveFileWithProgressA
14. MoveFileWithProgressW
15. OpenFile
16. ReadFile
17. ReadFileEx
18. ReadFileScatter
19. ReplaceFile
20. ReplaceFileA
21. ReplaceFileW
22. WriteFile
23. WriteFileEx
24. WriteFileGather

4. Results and Discussion

The test outcome could not avoid outliers. We defined outliers as a performance record in a particular round that extremely deviated from other rounds. On the assumption that the appearance of outliers is caused by the interruption of another program, when we encountered an outlier we simply reran the whole 30 rounds tests for that particular program again. This gives the cleanest data from which to determine the relative timing of the three different settings.

The outcome of the experiment was an interesting discovery. Regardless whether the computer had plain, antivirus or hook setting, all programs took a longer time during their first round of execution up to its GUI activation (See Table 1). However, a computer with antivirus and hook caused delay at the first round to become much longer. Also, in all cases, the antivirus caused more delay than hooking at the first round (See Table 2).

We conclude that computer does not really perform well when it runs a program at the first round upon reboot. Apparently the computer requires more time to launch that program. Assume the following scenario to help understanding this fact. Upon reboot, we ran MS Word. The software took more time to execute as compared to its subsequent execution. Without rebooting the computer, we then ran MS Excel. The program also experienced a delay during its first run. We also manually ran the software (without the help of autoit3 script) involving 30 rounds with 30 minutes interval between each round. It remains consistent that the first run took longer than the remaining 29 rounds for most programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Plain</th>
<th>Antivirus</th>
<th>Hook</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ms Word</td>
<td>1.71</td>
<td>4.88</td>
<td>4.65</td>
</tr>
<tr>
<td>Ms Excel</td>
<td>1.22</td>
<td>1.74</td>
<td>1.67</td>
</tr>
<tr>
<td>Ms Outlook</td>
<td>3.86</td>
<td>7.31</td>
<td>6.71</td>
</tr>
<tr>
<td>Firefox</td>
<td>3.19</td>
<td>4.89</td>
<td>4.09</td>
</tr>
<tr>
<td>IE</td>
<td>2.82</td>
<td>5.04</td>
<td>3.90</td>
</tr>
<tr>
<td>M.Player</td>
<td>1.27</td>
<td>3.12</td>
<td>1.49</td>
</tr>
<tr>
<td>Winamp</td>
<td>2.66</td>
<td>4.93</td>
<td>2.27</td>
</tr>
<tr>
<td>YM</td>
<td>2.30</td>
<td>4.36</td>
<td>2.84</td>
</tr>
<tr>
<td>LIVE</td>
<td>2.42</td>
<td>3.22</td>
<td>2.44</td>
</tr>
<tr>
<td>AOE2</td>
<td>2.19</td>
<td>2.76</td>
<td>2.26</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>AVG</th>
<th>Hook</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word</td>
<td>264.86%</td>
<td>271.44%</td>
</tr>
<tr>
<td>Excel</td>
<td>142.63%</td>
<td>136.91%</td>
</tr>
<tr>
<td>Outlook</td>
<td>189.49%</td>
<td>173.98%</td>
</tr>
<tr>
<td>Firefox</td>
<td>153.15%</td>
<td>128.05%</td>
</tr>
<tr>
<td>IE</td>
<td>178.48%</td>
<td>138.07%</td>
</tr>
<tr>
<td>M.Player</td>
<td>245.33%</td>
<td>116.83%</td>
</tr>
<tr>
<td>Winamp</td>
<td>185.12%</td>
<td>85.31%</td>
</tr>
<tr>
<td>YM</td>
<td>185.12%</td>
<td>106.57%</td>
</tr>
<tr>
<td>LIVE</td>
<td>1.33%</td>
<td>1.01%</td>
</tr>
<tr>
<td>AOE2</td>
<td>125.66%</td>
<td>103.11%</td>
</tr>
</tbody>
</table>

Exclusively, the tests on Winamp and IE showed that a delay in a computer with antivirus is more than the penalty occurred in a computer with hook throughout 30 rounds. Also, when we ran Winamp in computer with antivirus, it did not run at a flat speed. The same experience happened to AOE2 and YM with hooking. These facts can be seen from their coefficient of variation. Coefficient of variation is defined as the
ratio of the standard deviation to the mean. Although initially we assumed that they were outliers, we obtained similar outcomes when they were retested for several times.

We considered a plain computer as a base for benchmarking. The delay in a computer with antivirus and hook were measured in percentage relative to the base.

### Table 3: 30 rounds running variances (in seconds)

<table>
<thead>
<tr>
<th>App</th>
<th>Plain</th>
<th>Antivirus</th>
<th>Hook</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>Std Dev</td>
<td>Mean</td>
<td>Std Dev</td>
</tr>
<tr>
<td>Word</td>
<td>0.42</td>
<td>0.25</td>
<td>0.98</td>
</tr>
<tr>
<td>Excel</td>
<td>0.37</td>
<td>0.16</td>
<td>0.40</td>
</tr>
<tr>
<td>Outlook</td>
<td>0.84</td>
<td>0.57</td>
<td>1.33</td>
</tr>
<tr>
<td>Firefox</td>
<td>0.79</td>
<td>0.45</td>
<td>0.88</td>
</tr>
<tr>
<td>IE</td>
<td>0.78</td>
<td>0.39</td>
<td>1.89</td>
</tr>
<tr>
<td>M.Player</td>
<td>0.41</td>
<td>0.16</td>
<td>0.48</td>
</tr>
<tr>
<td>Winamp</td>
<td>0.56</td>
<td>0.40</td>
<td>2.95</td>
</tr>
<tr>
<td>YM</td>
<td>0.66</td>
<td>0.32</td>
<td>0.75</td>
</tr>
<tr>
<td>LIVE</td>
<td>0.61</td>
<td>0.34</td>
<td>0.66</td>
</tr>
<tr>
<td>AOE2</td>
<td>0.42</td>
<td>0.25</td>
<td>0.98</td>
</tr>
</tbody>
</table>

Throughout 30 rounds running, we observed the running performance of the computer with those three different settings (See Table 3). A plain computer and a computer that has the antivirus ran mostly at a consistent speed. However, a computer that has API hook program ran inconsistently. This can be observed in a higher number of its coefficient of variation while the computer with antivirus has a smaller value.

A similar test was conducted at a different computer and the same result patterns were still obtained. The machine has Intel Pentium 4 CPU 3.2 GHz HT processor with 512 MB RAM and 80 GB hard disk. We used Ms Virtual PC installed with Windows XP SP2 on top of existing Windows XP SP2 installed on the physical machine. It is clear that hook causes delay at all times, but is a bit better than antivirus at the first run. Figure 1 shows one of the outcomes conducted on MS Word.

The speeds between the three different computer settings were compared. Exclusively for the first run test, the plain computer ran the fastest followed by the computer with hook and computer with antivirus. Nevertheless, some programs performed slowest in API hook environment. The speed and performance of the machine used also contributed to the speed of the experimental process. The relevant point to note was their performance difference ratios.

The results obtained from the experiments suggest that API hook is a suitable option to look for malware detection without much worrying about performance penalty. Result obtained can be interpreted as the first run always slower that the subsequent execution. Also, the plain computer ran the fastest followed mostly by the computer with hook and computer with antivirus exclusively at first run.

The specific timings would be expected to vary on computers with different specifications. However, we have seen the same patterns concerning first vs subsequent executions, on all three test settings (plain, computer with antivirus and computer with API hook), across all repetitions, and on two different computers. Therefore we believe that these observations are true in general.

From our experiment using API hooking, we found that different programs had different API execution sequences. Some programs had a high number of a particular function invocation while others did not use it at all. This could be due to the variety in software features and the different programming language and algorithm used. Whatever the cause, it gives some support to the idea that API calls might be the basis of a signature for recognizing an application, and hence perhaps malware.

### 5. Conclusion

We are conducting research into whether looking for patterns in sequences of API calls can be an effective way of distinguishing between benign and malware code execution. As a step in this process, we have presented a technique for API hooking that avoids the high overhead associated with trapping all API calls by focusing on selected ones.

We conducted experiments into the performance of this technique. Our objective was to understand the relative performance of three scenarios based on running on a plain computer, a computer with antivirus software and a computer with an API hook program.
Results suggest that optimized API hooking does not impose much burden on the performance of a computer. The burden is similar to that imposed by the tested commercial antivirus software. We conclude from this that it is feasible to investigate API hooking for malware detection.

Our future work includes capturing API sequences generated by malware. We will profile those sequences and utilize the profile in our malware detection engine. We will optimize our matching algorithm and compare its performance against existing similar ones used in similar research. We proposed Malicious Code Detection Inspired by Human Immune System (MaCDI)[1] where two detection phases were introduced: Adolescent and Mature Phase. The first phase uses a malware profile matching mechanism, whereas the second phase uses a program profile instead.

6. References