Detecting the Code Injection by Hooking System Calls in Windows Kernel Mode

[1]Hung-Min Sun, [2]Yu-Tung Tseng, [3]Yue-Hsun Lin
[123]*Dept. of Computer Science, National Tsing Hua University, Taiwan*
[1]*hmsun@cs.nthu.edu.tw* , [23] *{se7en, tenma}@is.cs.nthu.edu.tw*

ABSTRACT

In present Microsoft Windows operating system, there are unofficial approaches to inject code into other running processes. We discuss the methods and corresponding potential threats in this paper. Malicious software may use these approaches to infect authorized processes to launch attacks inside the system even under the protection of antivirus and firewall software. After analyzing these runtime code injections, we proposed the mechanism – Detecting the Code Injection Engine (DCIE). DCIE is implemented as a loadable kernel-mode driver that is able to detect runtime code injections, and the maximal overhead caused by DCIE is less than 3.26%. The minor overhead makes DCIE suitable to be installed on Windows OS or combine with other software to increase system security.

1: INTRODUCTION

With the wild spread of virues, trojans, worms, malware, and spyware, most people protect their personal computer by antivirus and firewall software. However, on October 31, 2005, Mark Russinovich revealed the rootkit coming from a Sony music CD in his blog [9]. This rootkit will be secretly installed into Microsoft Windows operating system after inserting a Sony music CD into the computer. In fact, it is a kind of Digital Rights Management (DRM) software that provides protection against unauthorized copies of the CD, but it hides its process information and activity from being detected. This informs us that malicious software may use similar approaches to avoid the protection of antivirus software, and even the detection of firewall software.

Through analyzing present firewall software, most firewall software has at least two main components. The first component is the packet filter which is typically implemented as NDIS (Network Driver Interface Specification) drivers in order to inspect all inbound and outbound packets. According to the access control policy defined by user, it will examine the information in each packet such as protocol type, source/destination IP address, and port number to allow or deny the connections of the computer. The other one is the application filter which is also implemented as drivers running in kernel mode. It will check the authorized application list to perform the action when processes try to send or receive data. While an unauthorized process tries to request the network action, it will block the request and then notify the user of the request. After the user defining the rule, the application filter then permits or denies the action. The above two main components actually accomplish most functions to defense against external threats.

However, there still exist potential threats. In Windows operating system, it is considered a legitimate behavior that a process creates a remote thread in another running process. Thus an unauthorized application may inject malicious code into an authorized process and then execute it to bypass the application filter without causing a warning. This kind of attack seems like buffer overflow, but it does not require a bug to trigger the injected malicious code in the target process. In other words, malware, spyware, and rootkits can launch the attack inside the system to send data or open a backdoor silently even under the protection of firewall software.

In this paper, we discuss the methods how to inject code into running processes and corresponding potential threats. By analyzing these methods, we propose a detecting mechanism called Detecting the Code Injection Engine (DCIE) on the Microsoft Windows operating system. DCIE is implemented as a loadable kernel-mode driver that is able to dynamically monitor every process in the system and provide users with more precise information about the suspected injecting behavior.

The remainder of the paper is organized as follows. Section 2 describes the related research. Section 3 discusses runtime code injection and potential threats. Section 4 presents the DCIE mechanism. Section 5 presents experimental evaluation results. Section 6 summarizes the paper and discusses future work.

2: RELATED WORK

Intrusion detection systems developed to detect external threats have been quite successful. Although it is very little similar work for internal threats, there is still less work related to the area of internal threat detection. Internal threats represent a higher possibility of compromising the system since they are authorized to obtain system information and access the file system directly. Therefore, system call information is also heavily used to detect internal threats.

The work in [11] built profiles on the system call traces by monitoring file access and process activity. Then they used these profiles to detect insider misbehavior and buffer overflow attacks. There is another theory presented by Chinchani [4] et al. First, they described a model which records several aspects of insider threat, and afterward showed threat assessment to reveal possible attack strategies of an insider. In another work proposed by Liu [8] et al, they applied techniques

from external threat research to insider threats. The goal of their research is to empirically test the approaches of detecting external threat on malicious insider behavior.

However, most works about internal threat detection focus on Linux OS. We found an approach [14] that is implemented as a loadable Windows NT kernel module and is capable of selectively intercepting process creation requests. They proposed the execution management utility which is designed to prevent unknown software from executing without the approval of system administrator. Weidong et al. proposed another host based system, BINDER [16], to detect a large class of unknown malware. Their approach handled the problem of automated detection of break-ins caused by unknown malware by infer user intent. They correlated outbound connections with user-driven input at the process level under the assumption that user intent is implied by user-driven input.

3: RUNTIME CODE INJECTION

In this section, we present two methods how a process injects and executes codes in another process, and discuss the corresponding potential threats.

3.1: DLL INJECTION

This method is well documented in the article "Load Your 32-bit DLL into Another Process's Address Space Using INJLIB" by Jeffrey Richter [7]. Due to limited space, we only introduce the concept of this method in this section.

The main idea is to write the injected code as a Dynamic-Link Library (DLL) and force the target process to load the DLL. However, we can not alter the execution path of the existing threads in the target process. For this reason, we have to create a new thread in the target process and set the execution path of the thread to load the DLL.

Figure 1(a) illustrates the flowchart of the DLL injection. The first step is optional. We have to enable the debug security privilege if we want to inject a DLL into system processes. Then we retrieve the process handle of the target process for the following steps. In the third and fourth steps, we allocate memory in the target process and write the full path name of the DLL in the allocated memory. Finally, we create a remote thread in the target process and pass the address of *LoadLibrary* API as the thread routine. As a result, the new thread created in the fifth step will start to map the DLL into the address space of the target process.

3.2: BINARY CODE INJECTION

Another approach is similar to DLL injection. In binary code injection, we set the remote address of the injected binary code segment as the thread routine and create a remote thread to execute the injected binary code. Figure 1(b) illustrates the flowchart of binary code injection. Notice that this method injects into the target process without an additional DLL stored in the system. By considering the above steps, this method seems easier than DLL injection. However, the key point is the construction of the binary code.

The injected binary code can not be initialized as a DLL mapped into the address space of the target process. Basically, the binary code can be implemented as a function in the malicious process, but there are still problems. First of all, the binary code uses absolute addresses to reference variables or call functions, but these are the actual addresses in the address space of the malicious process. That is to say, i.e., the injected binary code can not retrieve the same data or call the same functions at these addresses in the target process. It would be less trouble to combine all subroutines into a large function unless you want to inject each subroutine and pass their remote addresses to the injected binary code. Secondly, the binary code is relocated in the malicious process. The target process will not relocate again the injected binary code. Mostly, we will crash the target process when we create a remote thread to execute the injected binary code. There are more implementation considerations and coding suggestions proposed by Robert Kuster [13].

With the exception of Robert Kuster's method, we also can use inline assembly to construct the binary code. Although this is much complicated to implement, we do not need to concern about the relocation of the injected binary code. Furthermore, the binary code can be much more under control and reliable if we use assembly language to implement the whole malicious process. This can refer to the rattle's document [12].

In a nutshell, binary code injection is much more complicated and riskier than DLL injection. On the other hand, it provides a flexibility to inject without an additional DLL.

3.3: POTENTIAL THREATS

Most firewall software for Windows OS allows outbound network communication according to the authorized application list. However, an unauthorized process may runtime inject codes into an authorized process to bypass the detection from firewall software. We list the potential threats as follows:

1. After injecting the DLL into an authorized process, the unauthorized process creates a remote thread to load the injected DLL. When the injected DLL initializes, it immediately connects to the attacker's computer.
2. The injected DLL does not connect out immediately. It only hooks the *connect* API to replace the original destination address with the IP address of the attacker's computer. Instead of

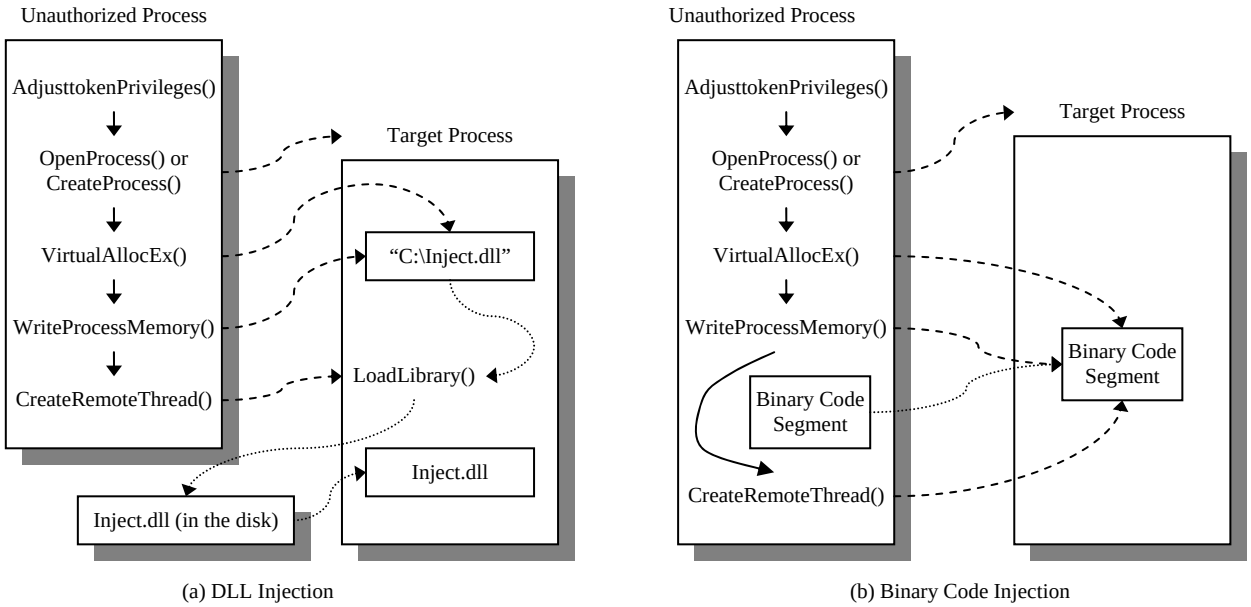

Figure 1. Runtime Code Injection

creating a new network connection, this threat only alters the destination address of the connecting socket and redirects it to the attacker's computer.

3. The injected DLL hooks the *accept* API to hijack the connecting socket by checking its source address coming from the attacker's computer instead of connecting out. This threat does not try to bypass the outbound detection, and it also does not listen to any port that would be caught by the packet filter of firewall software. In brief, this potential threat tries to hide in the injected authorized server application and hijack the connecting socket.
4. The last threat use binary code injection to infect authorized applications without an external DLL. We use assembly language to implement the injected code as the shellcode [2] which is a relocatable piece of machine code used as the payload in the exploitation of a software bug. However, in this threat, the authorized process does not need a bug to be overflowed. We directly create a remote thread to execute the injected shellcode.

In the second and the third threat, we use API hooking [10] [6].[6] It was originally developed for the debugging purpose to monitor activities and relationships of API functions in binary executable programs, especially components of operating system and third party software that can not access the related source codes. It also applied to extend the originally offered functionalities by intercept and substitute API functions. In other words, we can combine runtime code injection and API hooking to alter program execution for specific needs.

4: DETECTING THE CODE INJECTION ENGINE (DCIE)

Having discussing the runtime process injections and noticed the corresponding potential threats, we introduce our mechanism – Detecting the Code Injection Engine (DCIE).

4.1: HOOKING SYSTEM CALLS

According to the analysis of runtime code injections, we obtained that the unauthorized processes must call several critical APIs to complete the injection. These critical APIs are provided by Win32 subsystem. We must notice that the APIs provided by subsystem DLLs do not perform the functionalities, but validate parameters, update subsystem data structures into native data structures, and finally forward each request to corresponding native APIs provided by ntdll.dll. In other words, the APIs provided by subsystem DLLs are only the wrappers of native APIs in the lower level of Windows OS.

As a result, the process which directly calls native APIs, running in the Win32 subsystem, can skip the hooks to Win32 APIs. Of course, we can hook native APIs to achieve global system-wide hook just like we hooking Win32 APIs. However, native APIs provided by ntdll.dll are similar to Win32 APIs provided by subsystem DLLs. Both they are wrappers of the entry points to lower level system functions and the main job of native APIs is to dispatch the requests from subsystems to corresponding system calls provided by ntoskrnl.exe. Therefore, it is possible that the process running in the Win32 subsystem may directly call system calls by simulating native APIs, although it is a hard work fighting with assembly language. To avoid these situations, we have to hook system calls in kernel mode.

Most important of all, once we hook a system call, it is a system-wide hook even applies to device drivers. All requests to the system call will be intercepted by our hook.

Owing to monitor the entire OS and avoid the situation that processes may directly invoke system calls, we decided to hook system calls in kernel mode instead of APIs in user mode. In Windows OS, System Service Table (SST) [15] is an array in kernel mode which stores entry points of system calls. In other words, we can replace the entry point of system call with the address of our routine to hook system call and check its arguments.

After analyzing runtime code injections, we obtained that the injecting procedure can be divided into four stages. In the first stage, an unauthorized process must retrieve the handle of the target process via *OpenProcess* or *CreateProcess* APIs. Then it calls *VirtualAllocex* to allocate memory in the second stage. In the third stage, it subsequently calls *WriteProcessMemory* to write the full DLL name or binary code segment into the allocated memory. Finally, in the fourth stage, it calls *CreateRemoteThread* to force the target process to load the DLL or execute the injected binary code. As a result, we have to hook the corresponding system call of the API in each stage, and check its arguments stage by stage.

However, there is an exception in the first stage. An unauthorized process may call *DuplicateHandle* API to duplicate the handle of the target process to another unauthorized process. Although the first unauthorized process will be examined in the following stages, the second unauthorized process will not since it does not call OpenProcess or CreateProcess APIs. Therefore, we decided to hook the corresponding system calls of the following three stages. We replaced the entry point addresses of *ZwAllocateVirtualMemory*, *ZwWriteVirtualMemory*, and *ZwCreateThread* by checking the arguments of these three system calls to detect the behavior of runtime code injections.

4.2: IMPLEMENTATION

We divide DCIE into two parts. In the first part, we check the arguments of ZwAllocateVirtualMemory and ZwWriteVirtualMemory. These two system calls are used to allocate and write virtual memory in the user mode address range. However, it is quite frequent that a process calls these two system calls to operate its memory. As a result, it will significantly degrade the overall system performance if we check each request to these two system calls. The important point to note is that we only focus the operations between different processes.

In runtime process injections, an unauthorized process must allocate memory in the target process to write the full DLL path name or the binary code. Therefore, we can call *PsGetCurrentProcessId*, kernel API, to retrieve the process handle of the current process in the hooked system calls. Then we compare it with the process handle of the target process to recognize whether the system call request belongs to inter-process operation or not. If we find an inter-process operation in ZwAllocateVirtualMemory system call, we record the handle of the target process and the base address of allocated memory in the *suspect* list for another check. If we find an inter-process operation in ZwWriteVirtualMemory system call, we compare the handle of the target process and the writing base address in the suspect list. Once these two arguments match the record in the suspect list, we move the record into the *danger* list for the second part.

In the second part, we check the arguments of ZwCreateThread. We also only focus the system call requests between different processes. DLL injection will create a remote thread in the target process to load the DLL in the final step. Therefore, it must set the start address of the creating thread as the address of *LoadLibraryA* API in kernel32.dll. The same observation applies to binary code injection that it will create a remote thread with the start address which is allocated in the previous step. In other words, if we have recognized that the system call request belongs to inter-process operation, then we check the start address of the creating threads.

In ZwCreateThread system call, the *ThreadContext* parameter stores the initial processor context for the thread. The thread has two start addresses. One for kernel thread is stored in structure member *Eip*, and one for user-mode thread is stored in structure member *Eax*. Since the addresses of LoadLibraryA API and the allocated memory are in user mode, we only have to examine ThreadContext->Eax. Once the ThreadContext->Eax is set as the address of LoadLibraryA API, DCIE will immediately inform users of the behavior of DLL injection. If it matches the records in the danger list, DCIE will immediately inform user the behavior of binary code injection. In addition, if the target process terminates, the corresponding records in the suspect and danger list will be removed.

5: EVALUATION

In this section, we first evaluate the performance of DCIE. Then we analyze overhead caused by DCIE. Finally, we compare DCIE with other schemes.

5.1: PERFORMANCE

We used PCMark05 [5] to evaluate the performance of DCIE installed on Windows XP SP2 and measured average benchmarks. The CPU benchmarks were run on a desktop PC with an Intel 2.0GHz Celeron CPU and 448MB of RAM.

Table 1 shows the results of CPU benchmark in single thread test and Table 2 shows the results of CPU benchmark in multithread test. Compared with the benchmarks of the normal desktop PC, the maximal overhead caused by DCIE is less than 3.26%. These benchmarks show that DCIE is suitable to be installed on Windows OS to detect suspected code injections.

Benchmark	Normal	DCIE	Overhead (%)
File Compression (MB/s)	3.029	2.988	1.35
File Decompression (MB/s)	75.528	75.25	0.37
File Encryption (MB/s)	44.074	43.152	2.09
File Decryption (MB/s)	38.634	38.547	0.23
Image Decompression (MPixels/s)	14.466	14.105	2.50
Audio Compression (KB/s)	1371.476	1368.955	0.18

Table 1. Single Thread Performance Test

Benchmark	Normal	DCIE	Overhead (%)
File Compression (MB/s)	1.561	1.544	1.09
File Encryption (MB/s)	21.845	21.781	0.29
File Decompression (MB/s)	19.155	19.061	0.49
File Decryption (MB/s)	9.552	9.241	3.26
Image Decompression (MPixels/s)	3.612	3.593	0.53
Audio Compression (KB/s)	331.738	326.962	1.44

Table 2. Multithread Performance Test

Features	Outpost	ZoneAlarm	DCIE
Number of hooked system calls	1	2	3
Number of monitoring stages	1	1	3
The stage of blocking threats	3rd	1st	4th
Detecting all potential threats	No	Yes	Yes
Ability of identifying injection methods	Yes	No	Yes

Table 3. Comparisons with Other Systems

5.2: OVERHEAD ANALYSIS

In order to monitor all running processes in the OS, DCIE hooks three system calls in kernel mode to detect suspected code injections. Although it is quite frequent that a process allocates its memory to store data, DCIE will not incur significant degradation to overall system performance after installed it. Since DCIE only focuses the operations between different processes, it will not cost too much CPU time to check parameters of the hooked system calls.

In addition, it is not common that a process has to inject code into another process for its execution. Although there are situations that a process injects DLL into other processes to extend the functionalities of the hooked APIs, it is rare that a process inject binary code into other processes. Therefore, DCIE does not have many chances to incur system overhead.

5.3: Comparisons

In order to compare with other schemes, we implemented the proof of concept codes to represent the potential threats described in the section 3.3. The proof of concept codes use DLL injection and binary code injection to infect authorized processes such as web browser, ftp clients, instant message software and server applications. We tested commercial firewall software to check if they can detect runtime code injections and provide protection for the running process on the authorized application list. In our experiments, there are two firewall software, Outpost [1] and ZoneAlarm [17], can detect runtime code injections. We compare DCIE with these two firewall software and list the results in Table 3.

Considering the first and second feature, it is obvious that the detecting mechanisms of these three schemes are quite different. We used SDTrestore [3] to determine which system calls are hooked by Outpost and ZoneAlarm. Outpost hooks ZwWriteVritualMemory which is called in the third stage of runtime code injection. ZoneAlarm hooks ZwOpenProcess, ZwCreateProcess, and ZwCreateProcessEx which are called in the first stage. These two firewalls only monitor the specific stage of runtime code injection. On the contrary, DCIE hooks ZwAllocateVirtualMemory, ZwWriteVirtualMemory, and ZwCreateThread to monitor the last three stages.

Due to hook different system calls, each scheme detects and blocks runtime code injection in the different stage. We list the analysis of three schemes as the third feature since it deeply affects the fourth feature. The fourth feature presents the ability of identifying the method of runtime code injection. Since ZoneAlarm hooks system calls in the first stage, it can not distinguish the difference between DLL injection and binary code injection. Outpost hooks system call in the third stage. Once a process writes data into another process more than 16 bytes, Outpost will recognizes the action as manipulation and immediately terminates the injected process. Moreover, Outpost hooks LoadLibrary API of kernel32.dll in user mode to detect DLL injection. As to DCIE, we hook the last three stages and check the last system call. Therefore, DCIE can identify the methods of runtime code injections.

In the last feature, only ZoneAlarm and DCIE can detect all potential threats. There is an exception to Outpost. It permits the action that a process can creates a child process and then writes data into the child process. As a result, an unauthorized process can create an authorized process as its child process and then inject code into it to bypass the detection. In contrast, ZoneAlarm and DCIE can detect all potential threats. However, ZoneAlarm can not identify the methods of runtime code injections as DCIE.

6: CONCLUSIONS AND FUTURE WORK

In this paper, we introduce and discuss the potential threats caused by runtime code injections. By analyzing runtime code injections, we proposed the mechanism – Detecting the Code Injection Engine (DCIE) on Microsoft Windows Operation system.

DCIE is implemented as a loadable kernel-mode driver that is able to monitor every process in the system, and it does not impose too much overhead (less than 3.26%) to system. We believe that DCIE is suitable to be installed into Windows OS to provide users with more precise information about the suspected injecting behavior. Furthermore, DCIE also can be used to protect the running process in the authorized application list.

Although we can detect runtime code injections, we do not understand the intent of the injected DLL or binary code. In the future, we will improve our system in two aspects. In the first aspect, we will enhance DCIE to recognize the intent of the injected code or directly combine DCIE with antivirus scan engine. However, there may be conflicting situations when there is more than one application that uses the same concept to detect runtime code injections. It is another future work to develop a coordinate mechanism to handle situations that different processes hook the same system calls.

Acknowledgements

The authors wish to acknowledge the anonymous reviewers for valuable comments. This research was supported in part by the project of Water Resource Agency, Ministry of Economic Affairs, Taiwan, under contract no. MOEAWRA0950070.

REFERENCES

[1] Agnitum. Outpost Firewall Pro & Free. Available: http://www.agnitum.com/

[2] Aleph One, Smashing The Stack For Fun And Profit *Phrack*, vol.7 p49-0x14, Nov, 1996 Available: http://www.phrack.org/phrack/49/P49-14

[3] Chew Keong TAN. Defeating Kernel Native API Hookers by Direct Service dispatch Table Restoration. 2004.

[4] Chinchani, R., Iyer, A., Ngo, H. Q., and Upadhyaya, S. , "Towards a theory of insider threat assessment," *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pp.108-117, 2005.

[5] FutureMark. PCMark05. Available: http://futuremark.com/products/pcmark05/

[6] Holy_Father. Techniqs of hooking API functions on Windows. 2002. Available: http://www.hxdef.org

[7] Jeffrey Richter, Load Your 32-bit DLL into Another Process's Address Space Using INJLIB *Microsoft Systems Journal*, vol.9 Number 5, May, 1994.

[8] Liu, A., Martin, C., Hetherington, T., and Matzner, S., "A comparison of system call feature representations for insider threat detection," *Systems, Man and Cybernetics (SMC) Information Assurance Workshop, 2005. Proceedings from the Sixth Annual IEEE*, pp. 340-347, 2005.

[9] Mark Russinovich. Sony, Rootkits and Digital Rights Management Gone Too Far. 2005. Available: http://www.sysinternals.com/blog/2005/10/sony-rootkits-and-digital-rights.html

[10] Matt Pietrek, Learn System-Level Win32 Coding Techniques by Writing and API Spy Program *Microsoft Systems Journal*, vol.9 Number 12, Dec, 1994.

[11] Nguyen, N., Reiher, P., and Kuenning, G. H., "Detecting insider threats by monitoring system call activity," *Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society*, pp. 45-52, 2003.

[12] rattle, Using Process Infection to Bypass Windows Software Firewalls *Phrack*, vol. 11 p62-0x0d, Jul, 2004.

[13] Robert Kuster. Three Ways to Inject Your Code into Another Process. 2003. Available: http://www.codeproject.com/threads/winspy.asp

[14] Schmid, M., Hill, F., Ghosh, A. K., and Bloch, J. T., "Preventing the execution of unauthorized Win32 applications," *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX '01. Proceedings*, pp. 175-183 vol.2, 2001.

[15] Sven B. Schreiber. *Undocumented Windows 2000 secrets : a programmer's cookbook* , Boston: Addison-Wesley, 2001.

[16] Weidong Cui., Randy H. Katz, and Wai-tian Tan, "Design and Implementation of an Extrusion-based Break-In Detector for Personal Computers," *21st Annual Computer Security Applications Conference (ACSAC'05)*, pp. 361-370, 2005.

[17] Zone Labs. Zone Alarm Firewall Pro & Free. Available: http://www.zonelabs.com/store/content/home.jsp