CHARACTERISTICS OF MALICIOUS DLLS IN WINDOWS MEMORY

Dae Glendowne, Cody Miller, Wesley Mcgrew, David Dampier

▶ To cite this version:

Dae Glendowne, Cody Miller, Wesley Mcgrew, David Dampier. CHARACTERISTICS OF MALICIOUS DLLS IN WINDOWS MEMORY. 11th IFIP International Conference on Digital Forensics (DF), Jan 2015, Orlando, FL, United States. pp.149-161, 10.1007/978-3-319-24123-4_9 . hal-01449075

HAL Id: hal-01449075
https://inria.hal.science/hal-01449075v1

Submitted on 30 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Chapter 9

CHARACTERISTICS OF MALICIOUS DLLS IN WINDOWS MEMORY

Dae Glendowne, Cody Miller, Wesley McGrew and David Dampier

Abstract Dynamic link library (DLL) injection is a method of forcing a running process to load a DLL into its address space. Malware authors use DLL injection to hide their code while it executes on a system. Due to the large number and variety of DLLs in modern Windows systems, distinguishing a malicious DLL from a legitimate DLL in an arbitrary process is non-trivial and often requires the use of previously-established indicators of compromise. Additionally, the DLLs loaded in a process naturally fluctuate over time, adding to the difficulty of identifying malicious DLLs. Machine learning has been shown to be a viable approach for classifying malicious software, but it has not as yet been applied to malware in memory images. In order to identify the behavior of malicious DLLs that were injected into processes, 33,160 Windows 7 x86 memory images were generated from a set of malware samples obtained from VirusShare. DLL artifacts were extracted from the memory images and analyzed to identify behavioral patterns of malicious and legitimate DLLs. These patterns highlight features of DLLs that can be applied as heuristics to help identify malicious injected DLLs in Windows 7 memory. They also establish that machine learning is a viable approach for classifying injected DLLs in Windows memory.

Keywords: Malware, DLL injection, memory analysis

1. Introduction

Malware manifests itself in a variety of forms in Windows systems depending on the malware authors' needs and capabilities. For example, malware may run as its own process, as code injected into another process, as a service or as a driver. New techniques for executing malicious software in Windows systems arise occasionally, designed by malware authors in order to subvert detection and analysis. Each form of mal-

ware has its own characteristics that distinguish it from others and each generates a different set of artifacts that may be used for detection.

Code injection describes any situation where malicious software might copy code to the memory space of an existing legitimate process, with the goal of executing the code either immediately or as part of a "hook" placed in the target process. Dynamic link library (DLL) injection is a form of code injection that inserts a malicious DLL into a separate legitimate process [7]. The DLL may be loaded directly at runtime or it could be loaded automatically the next time a process executes. This chapter focuses on malware that performs DLL injection to accomplish its goals.

Malware authors implement DLL injection for two primary reasons. First, injection provides a level of stealth to malware code as it executes; any actions taken by the DLL appear to originate from the process. Second, it grants malware the execution context of the "container" process. This allows malware to utilize system resources with the privileges of the host process such as the filesystem, registry and network access. If a botnet client needs to contact its command and control server, it may inject a DLL into a web browser to bypass host-level, process-specific firewalls that might otherwise block access.

This chapter explores the characteristics of malicious DLLs in Windows 7 x86 memory images. This is accomplished by executing malware samples in a sandboxed environment and acquiring memory. Volatility is used to extract features associated with a DLL sample, such as the host process, load path, base address and load count [16]. These features are analyzed to identify behavioral patterns in injected DLLs. Identifying these patterns serves two purposes. First, the patterns contribute to the general knowledge of malicious behavior in memory and can be used by forensic examiners as heuristics to aid in identifying malicious injected DLLs in Windows 7 memory images. Second, the patterns define characteristics of malicious injected DLLs that can help distinguish between malicious and legitimate DLLs. Distinctive behavioral patterns are necessary for machine learning to produce a robust model for reliably classifying new data. Machine learning has previously been used to classify malicious PE (portable executable) files using static and dynamic features [11]. Preliminary analysis reveals that it is also viable for classifying malware in Windows memory images.

This research makes two principal contributions. The first is a procedure for generating and processing large quantities of infected memory images to identify injected DLLs. The second contribution is the extraction of behavioral information drawn from the data as it relates to malicious and legitimate DLLs in a system.

2. Motivation

Several challenges are associated with identifying the use of malicious DLLs by a process. The use of legitimate DLLs by a process may make it more difficult to identify malicious DLL use. The variety of techniques used to inject malicious code, including DLLs, further increases the difficulty. This research address the complex and dynamic nature of DLL usage to identify DLLs in memory images that are considered to be malicious.

Runtime dynamic linking occurs when a DLL is loaded at runtime by a process or another DLL. A DLL loaded in this manner provides some required functionality and may be unloaded shortly after the requirement is met or it may persist within the process. Legitimate software that provides plug-in or add-on interfaces frequently uses this capability [12]. This leads to discrepancies in the DLLs loaded by a process based on the point in time at which its DLL list is examined.

Malware may be injected as a DLL into a running process by utilizing the same system code that is used to load legitimate runtime DLLs. The use of an identical loading mechanism enables malicious DLLs to better blend in with their legitimate counterparts. Many of the artifacts generated by this loading process are the same as, or at least not significantly different from, other legitimate DLL loading artifacts.

3. Related Work

While this work focuses on the effective identification of DLL injection events during dynamic analysis, a body of research covers the more general problem of detecting malware in memory. The Blacksheep rootkit detector [2], for example, compares memory images from multiple systems to perform a variety of analyses. Blacksheep compares the loaded kernel modules between memory images as well as the code in the kernel space itself. Kernel data structures and entry points to kernel code are also checked for differences. After the comparison procedure, memory images are clustered based on their combined distances from each other based on the comparison features.

Mandiant's Memoryze [8], when incorporated as a part of the Redline investigation tool, can be used to scan memory images and apply heuristics to assign scores to objects in memory that could be malicious. Other commercial products, such as Malwarebytes [1], scan memory in search of malicious software. However, the mechanisms and criteria used by these products are not always transparent and little information is published on the techniques that determine the features used to cluster and identify malicious code.

Gionta et al. [5] have designed an architecture for memory virus scanning as a service. The architecture allows for the efficient acquisition and scanning of memory in massive virtual environments. In this way, in-depth memory analysis techniques, such as those described in this chapter, can be efficiently used with minimal impact on virtual machine operations.

In the general case, determining if a piece of software would exhibit malicious behavior is undecidable [3]; extensive reverse engineering is typically required to understand the functionality of the underlying code and craft detection signatures. Signature-based scanners are grossly inadequate for detecting sophisticated modern malware [6]. This has led to the use of machine learning for classifying software as benign or malicious [11]. The research in this area is extensive, but the focus is normally on identifying features in a Windows PE file such as byte n-grams, API calls and opcodes, and using the features to train a classifier. The features listed in this work are based on in-memory structures and the way that observed malware uses the structures. However, there is some overlap with existing research because features are extracted from PE headers in memory.

4. Test Data Generation

The test data was generated from a set of 33,160 malware samples obtained in 2012 from VirusShare [14]. The malware samples were subsequently processed by VirusTotal [15]: 97.29% of the samples were labeled as malicious by ten or more scanners. Additional verification was done using Metascan [10], which labeled 97.16% of the samples as malicious with a minimum detection rate of ten scanners.

The malware samples were then processed using the Cuckoo Sandbox [4]. The samples were submitted to a SQLite database where they were queued until they were processed. The Cuckoo host sent each queued sample to a Cuckoo analysis virtual machine running Windows 7 x86 with 512 MB of allocated memory. Since the sandbox does not provide external access to the Internet, INetSim was used to simulate various services; this caused some malware to exhibit additional functionality. The malware was executed in the Cuckoo analysis virtual machine for four minutes while being monitored by the Cuckoo agent, which runs as a Python script in the Cuckoo analysis virtual machine. The data collected about the malware included Windows API calls; network, registry and file interactions; and a memory capture of the system after malware execution. After this data was collected and compressed on disk, the Cuckoo analysis virtual machine was restored to the baseline

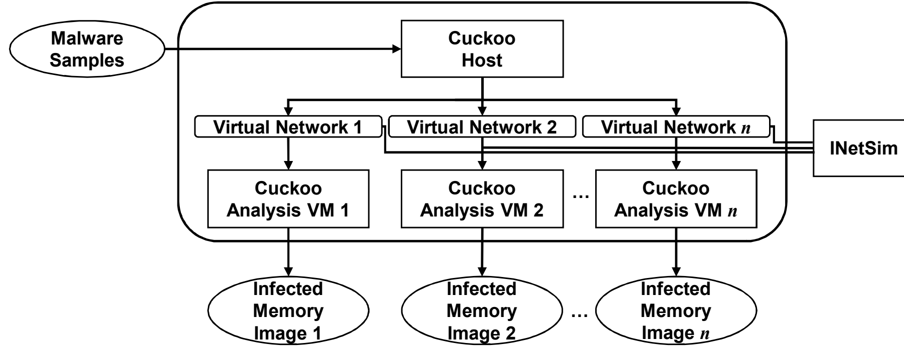

Figure 1. Data generation process.

snapshot and the next piece of malware from the queue was passed to the virtual machine. Figure 1 illustrates the process. Note that the data collected from Cuckoo by processing a piece of malware is referred to as a "cuckoo sample." A total of 33,160 cuckoo samples were generated.

While each cuckoo sample contains several types of information related to the execution of a piece of malware, this work focuses on data obtained solely from infected memory images. Volatility [16] was used to extract DLL artifacts from each memory image. The virtual address descriptor tree was traversed for each process to find nodes containing the mapped files. Artifacts were extracted from data structures associated with each node, including _LDR_DATA_TABLE_ENTRY, _MMVAD and _EPROCESS. The artifacts were combined to create a data point describing a given DLL.

5. Data Classification

Each memory image in the dataset had between 615 and 1,645 loaded DLLs. Each of these DLL data points had to be classified before analysis could be performed. A DLL and its host process were classified into one of four categories:

- Legitimate processes containing legitimate DLLs.
- Legitimate processes containing malicious DLLs.
- Malicious processes containing legitimate DLLs.
- Malicious processes containing malicious DLLs.

A whitelist of all the files was created from the baseline snapshot of the analysis virtual machine. For a given data point, the process and

Table 1. Dataset distribution.

	All	**Unique**
Injected DLLs	2,385	1,168
Legitimate DLLs	162,567	152,883

DLL were compared against the whitelist to determine the classification. Injected DLLs refers to DLLs categorized as "Legitimate processes containing malicious DLLs." Legitimate DLLs refers to either "Legitimate processes containing legitimate DLLs" or "Malicious processes containing legitimate DLLs." This work has opted to use the latter category because it provides a greater variety of legitimate DLLs than the former.

Each memory image contained the same legitimate processes. The processes tended to load the same DLLs across memory images, so using the set of "Legitimate processes containing legitimate DLLs" yielded large sets of repeated DLLs. When a malicious process is executed, it typically loads several system DLLs it requires for execution. These DLLs may not have been loaded by any of the other legitimate processes. Because the focus is on the DLLs and their behavior and appearance in memory, as long as they are legitimate, the nature of the loading process (whether malicious or legitimate) does not affect the analysis.

6. Injected DLL Characteristics

This section discusses the various characteristics identified in the injected DLLs. The characteristics drawn from the set of injected DLLs were contrasted with those of legitimate DLLs where appropriate. As mentioned above, 33,160 cuckoo samples were generated for the study; 955 of the samples exhibited DLL injection behavior. It is plausible that more samples than the 955 identified performed DLL injection, but they did not in this instance due to the lack of a required resource (e.g., configuration file, Internet connection or installed software).

The analyzed data was split into two subsets. The first subset contained all the injected DLLs and all the legitimate DLLs, 2,385 and 162,567 DLLs, respectively. The second subset contained the unique injected and legitimate DLLs for a given memory image, 1,168 and 152,883 DLLs, respectively. Table 1 shows the dataset distribution.

Target Processes. Malware that injects DLLs must specify a target process to host the malicious code. Figure 2 shows the target processes. In the dataset used in the study, some processes are more common than others. The most common processes targeted for injection

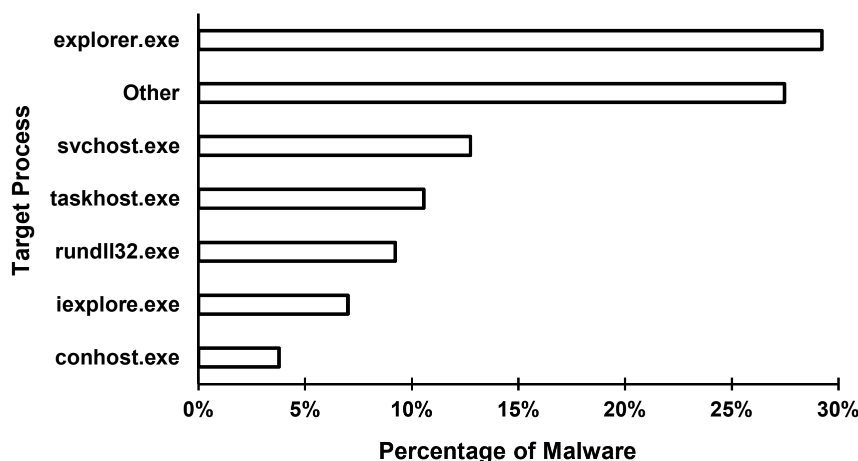

Figure 2. Target processes.

were `explorer.exe`, `svchost.exe` and `taskhost.exe`, accounting for more than 52% of injections. Each of these processes always runs on a Windows system and presents a large and/or varied set of DLLs at runtime. For example, `explorer.exe` had an average of 210 legitimate DLLs loaded at one time, and several instances of `svchost.exe` and `taskhost.exe` were typically in execution.

Number of Injections. A malware sample can choose to inject a DLL into any running process. Injecting DLLs into multiple processes can provide malware with greater survivability and versatility, but it also increases the chances of detection. In the dataset, 955 malware samples injected a DLL into a process. Figure 3 shows the number of injections per malware sample. Sixty percent (573) of the 955 samples targeted a single process while the remaining 40% (382) targeted two or more processes.

Simultaneous Loads. When malware injects a DLL into several processes, it often iterates through the active processes and injects the DLL as it finds its target(s). For malware that loaded into multiple processes, the load time extracted from _LDR_DATA_TABLE_ENTRY was examined to see how many DLLs had approximately the same load time. For a given DLL within a memory image, the DLL load time was compared against its load time in all the other processes containing the DLL. For every load time within one second, the simultaneous load value of the DLL was incremented by one (1). If the DLL appeared in multiple processes, but did not share a load time, then a value of zero (0) was

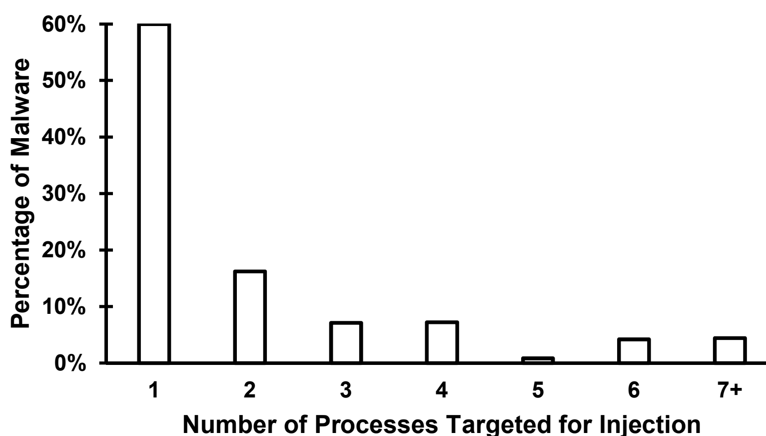

Figure 3. Number of injections per malware sample.

assigned. If a DLL existed only once within an entire memory image, a value of minus one (-1) was assigned. Of the DLLs that existed in more than one process (those with a value of zero (0) or greater), the number of DLLs detected with at least one simultaneous load for injected DLLs was 73.3%. For legitimate DLLs, the corresponding percentage was 45.4%. This shows that malicious DLLs tend to have approximately the same load times whereas the load times of legitimate DLLs are more varied.

Load Position. The *InLoadOrderModuleList* is a doubly linked list of _LDR_DATA_TABLE_ENTRY structures. The list is ordered based on when a DLL was loaded into a process, with the executable occupying the first position. The beginning entries are occupied by dynamically linked DLLs named in the import address table. DLLs loaded at runtime naturally appear at the end of the list. Some loaded DLLs are volatile in that they are loaded and unloaded repeatedly during the lifetime of a process while others are more stable, remaining loaded for longer periods of time.

The load position was calculated for each DLL that existed in the *InLoadOrderModuleList*. The average load position was calculated for injected and legitimate DLLs. The average load position for all the injected DLLs was 83.7. The average load position for legitimate DLLs was 52.6. Note that the system ran for a short period of time, which may have affected the reliability of the results. Depending on the number of DLLs unloaded by a process, an injected DLL may appear closer to the beginning of the list.

Init Position. Similar to the *InLoadOrderModuleList*, the *InInitializationOrderModuleList* represents the order in which the `DLLMain` function of a DLL was executed. The average init position for all the injected DLLs was 87.4. The average init position for legitimate DLLs was 51.3. This result may also be affected by the short execution time of the analysis system.

Base Address. The base address is the virtual address in a process where a DLL is loaded. The default base address for DLLs is `0x10000000`. Since a process normally contains multiple DLLs and only one DLL can occupy a given virtual address within a process, many DLLs contain a `.reloc` section in the PE header that specifies how to translate its offsets. In all, 48% of the unique injected DLLs were loaded at the virtual address `0x10000000`; this is in sharp contrast to legitimate system DLLs, for which 99.99% of the DLLs were loaded at an address other than `0x10000000`.

Exported Function Count. The number of functions exported by each DLL were extracted and used to calculate the means and modes for the injected and legitimate DLLs. The injected DLLs exported considerably fewer functions on the average, with a mode of 2 and a mean of 13. Legitimate DLLs had a mode of 11 and a mean of 368.

Imported Function Count. The number of functions imported by each DLL were extracted and used to calculate the means and modes for the injected and legitimate DLLs. The mode of each type of DLL was similar with injected DLLs and legitimate DLLs having modes of 213 and 198, respectively. The means for injected DLLs and legitimate DLLs were 115 and 257, respectively.

Loaded from Temp. A common heuristic when looking for malware is searching binaries loaded from a temporary directory such as *%TEMP%* (*C:\Users\UserName\AppData\Local\Temp*). In all, 20% of the unique injected DLLs were loaded from a directory with *temp* in the path. Nearly all of these were from *%TEMP%*, but a small number were from the directories *C:\temp* or *C:\Windows\temp*.

Load Paths. Figure 4 shows the most common load paths for injected legitimate DLLs. In the case of legitimate DLLs, 97% were loaded from *C:\Windows\System32* or *C:\Windows\System32\en-us*.

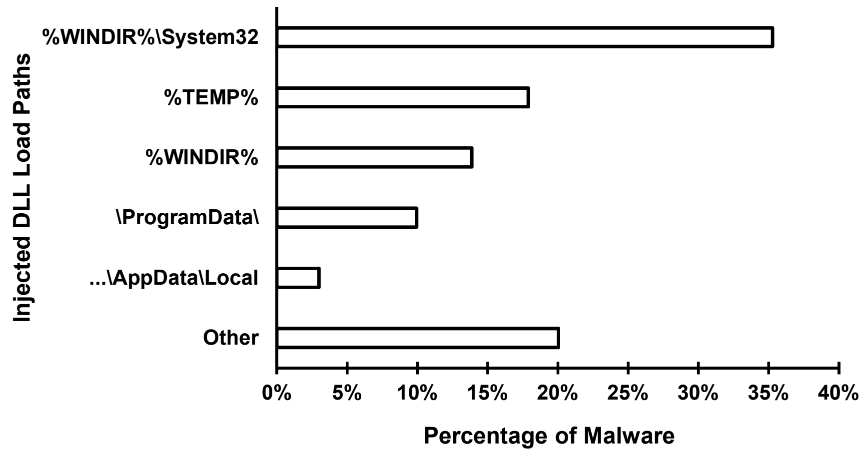

Figure 4. Malicious load paths.

COM Server. The Component Object Model (COM) is an interface standard used in the Windows operating system. It enables software to call code hosted by other software components without in-depth knowledge of its implementation. The calling component is the client and the hosting component is the server. Malware writers sometimes leverage the COM infrastructure to implement malicious code [13]. A COM server is required to export at least two Windows API functions: `DllGetClassObject` [9] and `DllCanUnloadNow` [9]. If these two API calls are seen in the exports of a DLL, then it is considered to be a COM server. Only 5.2% of the unique injected DLLs were implemented as COM servers.

COM Client. Windows binaries can call COM objects as clients. In order to use COM objects, the binary must call the `OleInitialize` [9] or `CoInitializeEx` [9] functions. DLLs importing either of these functions were considered to be COM clients. Only 2.1% of the unique injected DLLs in the dataset were capable of calling COM objects.

7. Threats to Validity

Certain issues have to be accounted for in the generation process. The analysis described above was restricted to a specific Windows operating system version on a single architecture. While all versions of Windows share the same general linking and loading procedures, the details of the mechanism can change over time and across architectures. In the case of "user-land" malware, the details of the mechanism may not impact

the results for newer versions of the operating system, but the behavior of malware that operates with elevated privileges may vary. This may be countered in future work by performing the data gathering and processing on multiple virtual machines with different operating system versions. The manner in which the resulting data is stored may have to be restructured to account for multiple DLL datasets for the different circumstances under which malware was executed.

Another issue is that the behavior of malicious software (and non-malicious software in the sandbox virtual machine) may vary based on the availability and status of the environmental resources and settings. Malware may choose not to act unless it determines that it can establish a "real" connection to a command and control server across the public Internet. It may choose not to inject DLLs into other processes unless it can detect the presence of email servers, active directory services or network shares. This would impact the ability to gather adequate DLL data, although it could be resolved by improving the fidelity and capability of virtual sandbox environments.

No additional software was installed on the system beyond the base installation. The legitimate DLLs referenced in this work are all Windows system DLLs. The behavior of third-party application DLLs may differ from that of the Windows system DLLs.

8. Conclusions

The research described in this chapter generated Windows 7 x86 memory images for 33,160 malware samples obtained from VirusShare. The malware samples were executed in Cuckoo Sandbox, a sandboxed dynamic analysis environment. The memory images were processed using Volatility to extract several artifacts associated with DLLs. DLLs that had been injected into legitimate processes were identified. From among the 33,160 cuckoo samples that were generated, 955 samples injected a total of 2,385 (1,168 unique) DLLs into legitimate processes. There were also 162,567 (152,883 unique) legitimate DLLs in the dataset. Analysis of this data revealed several characteristics of malicious injected DLLs and legitimate Windows DLLs. The characteristics contribute to the understanding of malicious DLLs and can be applied as heuristics to assist in identifying malware in Windows memory images. Additionally, they demonstrate the applicability of machine learning to the identification of malicious DLLs in Windows memory images.

Future research will examine other forms of in-memory malware such as processes and drivers. The research will employ a similar methodology to identify characteristics that can assist in distinguishing malicious

processes and drivers from their benign counterparts. The characteristics presented in this work will be used to build a feature set for training a machine learning classifier to identify malicious DLLs in memory. Feature selection algorithms will be applied to determine the most useful features and the resulting feature set will be evaluated using a variety of algorithms.

Acknowledgement

The authors wish to thank Puntitra Sawadpong for her assistance in formatting and proofreading this chapter.

References

[1] P. Arntz, Memory scan, Malwarebytes Unpacked, Official Security Blog, Malwarebytes, San Jose, California (blog.malwarebytes.org/development/2014/03/memory-scan), March 21, 2014.

[2] A. Bianchi, Y. Shoshitaishvili, C. Kruegel and G. Vigna, Blacksheep: Detecting compromised hosts in homogeneous crowds, *Proceedings of the ACM Conference on Computer and Communications Security*, 341–352, 2012.

[3] M. Christodorescu, S. Jha, S. Seshia, D. Song and R. Bryant, Semantics-aware malware detection, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 32–46, 2005.

[4] Cuckoo Foundation, Cuckoo Sandbox 1.2 (www.cuckoosandbox.org), 2014.

[5] J. Gionta, A. Azab, W. Enck, P. Ning and X. Zhang, SEER: Practical memory virus scanning as a service, *Proceedings of the Thirtieth Annual Computer Security Applications Conference*, pp. 186–195, 2014.

[6] D. Goodin, Antivirus pioneer Symantec declares AV “dead” and “doomed to failure,” *Ars Technica*, May 5, 2014.

[7] M. Hale Ligh, A. Case, J. Levy and A. Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux and Mac Memory*, John Wiley and Sons, Indianapolis, Indiana, 2014.

[8] Mandiant, Memoryze: Find evil in live memory, Alexandria, Virginia (www.mandiant.com/resources/download/memoryze), 2013.

[9] Microsoft, Windows API and Reference Catalog, Redmond, Washington (msdn.microsoft.com/en-us/library/ms123401.aspx).

[10] OPSWAT, Metascan Online, San Francisco, California (www.metascan-online.com/en).

[11] A. Shabtai, R. Moskovitch, Y. Elovici and C. Glezer, Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey, *Information Security Technical Report*, vol. 14(1), pp. 16–29, 2009.

[12] J. Shewmaker, Analyzing DLL injection, presented at the *GSM Conference*, 2006.

[13] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, No Starch Press, San Francisco, California, 2012.

[14] VirusShare, VirusShare.com – Because sharing is caring (`www.virusshare.com`).

[15] VirusTotal, VirusTotal (`www.virustotal.com`).

[16] Volatility Foundation, Volatility (`github.com/volatilityfoundation`).