Digital Investigation 28 (2019) S95–S104

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

Characteristics and detectability of Windows auto-start extensibility points in memory forensics

Daniel Uroz, Ricardo J. Rodríguez*

Centro Universitario de La Defensa, Academia General Militar, Zaragoza, Spain

ARTICLE INFO

Article history:

Keywords:
Memory forensics
System persistence
Malware
Auto-start extensibility points
Windows registry
Volatility

ABSTRACT

Computer forensics is performed during a security incident response process on disk devices or on the memory of the compromised system. The latter case, known as *memory forensics*, consists in dumping the memory to a file and analyzing it with the appropriate tools. Many security incidents are caused by malware that targets and persists as long as possible in a Windows system within an organization. The persistence is achieved using *Auto-Start Extensibility Points* (ASEPs), the subset of OS and application extensibility points that allow a program to auto-start without any explicit user invocation. In this paper, we propose a taxonomy of the Windows ASEPs, considering the features that are used or abused by malware to achieve persistence. This taxonomy splits into four categories: system persistence mechanisms, program loader abuse, application abuse, and system behavior abuse. We detail the characteristics of each extensibility point (namely, write permissions, execution privileges, detectability in memory forensics, freshness of system requirements, and execution and configuration scopes). Many of these ASEPs rely on the Windows Registry. We also introduce the tool `Winesap`, a Volatility plugin that analyzes the registry-based Windows ASEPs in a memory dump. Furthermore, we state the order of execution of some of these registry-based extensibility points and evaluate the effectiveness of our tool in memory dumps taken from a Windows OS where extensibility points were used. `Winesap` was successful in marking all the registry-based Windows ASEPs as suspicious registry keys.

Introduction

When a computer security incident occurs, an investigator (or better, a team of investigators) starts to examine the reasons behind the incident. The set of procedures employed is part of the *Incident Response* process. This process involves figuring out what happened and preserving information relating to the incident, asking the well-known 6 W's (*what*, *who*, *why*, *how*, *when*, and *where*) and ending with a report of the security incident stating lessons learned.

The incident response process is composed of different stages, as defined by NIST (Cichonski et al., 2012): (1) preparation; (2) detection and analysis; (3) containment, eradication, and recovery; and (4) post-incident activity (lessons learned and reporting). *Computer and network forensics* is one of the fundamental steps performed during the detection and analysis stage. In this regard, NIST also provides a guideline to integrate forensics techniques into incident response (Granc et al., 2006).

In the case of a security incident related to a compromised machine within an organization, an investigator can identify unauthorized and anomalous activity on the target computer or server, analyzing both the device drive and the memory. The latter case is named *memory forensics* (Ligh et al., 2014). In this paper, we focus on memory forensics since there are situations in which access to device drives are difficult (e.g., in cloud computing). Furthermore, the initial triage is faster since the data set in the memory is smaller than in the device drive.

Memory forensics is usually carried out by executing special software that captures and dumps into disk the current state of the system's memory as a snapshot file, also known as a *memory dump*. This file can then be taken offsite and analyzed with dedicated software such as Volatility (Walters and Petroni, 2007) to search for evidences of the incident.

A memory dump is full of data to analyze. Every element susceptible to analyze is termed as a *memory artifact*. A memory dump contains a snapshot of the running processes, as well as other system information such as logged users, open files, or open network connections at the time of capturing the dump. Furthermore, since the memory used by object resources is not usually zeroed out when freed, the memory dump also contains this kind of

* Corresponding author.
E-mail address: rjrodriguez@unizar.es (R.J. Rodríguez).

https://doi.org/10.1016/j.diin.2019.01.026

data (Ligh et al., 2014). Hence, any memory artifact can be retrieved from a memory dump either using the appropriate internal OS structures to go through the data content or using a pattern-like search in the full dump.

Nowadays, one of the most common security incidents is the presence of software specially designed with malicious purposes (known as malicious software or *malware*). The life cycle of malware is composed of several stages, which are similar to the stages of an Advanced Persistent Threat (APT) (Sood et al., 2013): first, it enters into the target computer to compromise it (using drive-by downloads, spear phishing, or other social engineering techniques, for instance). Then, the malware makes itself persistent in the system, i.e., it uses a persistence strategy to ensure that it will persist in the system within system reboots. Finally, it carries out its nefarious purposes.

The persistence stage of the malware is mainly motivated by the cybercriminal motto "*the longer the system is infected, the more the revenue*". The techniques used to persist in the system have been named *Auto-Start Extensibility Points* (ASEPs) in the literature (Wang et al., 2004; Russinovich and Margosis, 2016). More formally, ASEPs refer to the subset of OS and application extensibility points that allow a program to auto-start without any explicit user invocation. Note that these extensibility points are also used by legitimate programs, such as system services or update agents, among other software.

In this paper, we focus on Windows OS since it is still the most predominant target platform of malware attacks, according to recent statistics (AV-TEST Institute, 2018). In this regard, we study all the extensibility points in Windows OS that are susceptible to be used or abused by malware so that it can persist in the system. Furthermore, we also determine the characteristics of each extensibility point, such as the user privileges needed by the malware to persist, the execution privileges of the launched program, its detection in memory forensics, or its execution and configuration scope.

The contribution of this paper is two-fold: (i) we provide a taxonomy of ASEPs of Windows OS, defining four different categories: *system persistence mechanisms*, *program loader abuse*, *application abuse*, and *system behavior abuse*. Every category is further divided into different persistence techniques, detailing also their characteristics; (ii) we introduce the tool `Winesap`, which extends the Volatility framework and allows a memory forensic analyst to detect the presence of unknown and rare programs in registry-based ASEPs. Regarding evaluation, we have first empirically tested the order of execution of the registry-based Windows ASEPs that launch programs with every signed-in user. Then, we have evaluated the effectiveness of our tool for detecting the presence of Windows ASEPs in different memory dumps from a virtual machine. `Winesap` succeeds in detecting all registry-based Windows ASEPs, marking them as suspicious registry keys.

This paper is organized as follows. Section 2 describes the previous concepts needed to follow the rest of this paper (in particular, the Volatility framework and the Windows Registry). The taxonomy of Windows ASEPs proposed in this paper is introduced in Section 3. We also detail the characteristics of each ASEP. Section 4 presents `Winesap` and the experiments carried out to verify the tracking down of registry-based Windows ASEPs in memory forensics. Related work is discussed in Section 5. Finally, Section 6 draws conclusions and states future work.

Previous concepts

In this section, we first introduce the Volatility framework on which our tool `Winesap` relies. Then we briefly describe the Windows Registry and give some details of its analysis with Volatility.

The Volatility framework

The Volatility framework (Volatility for short) was released in 2007 at the BlackHat DC conference (initially called *Volatools*) (Walters and Petroni, 2007) as a result of years of published academic research into advanced memory analysis and forensics. Since then, Volatility has become a *de facto* standard for analyzing memory dumps in computer forensics. Volatility is an open source project released under GNU GPLv2 and currently maintained by *The Volatility Foundation*, a non-profit organization that promotes open source memory forensics.

Volatility supports the analysis of memory dumps from Windows, Linux, and Mac OS, in both 32-bit and 64-bit environments. Implemented in Python, the Volatility framework provides a memory forensic analyst with an open collection of tools to extract digital artifacts from those memory dumps.

Apart from the tools that are already distributed with Volatility, it provides developers with a rich, scriptable API to implement their own plugins or even to extend the features of the official tools. Furthermore, every year *The Volatility Foundation* organizes a contest to incorporate new features in the tool.

Some examples of community contributions in recent years are ProcessFuzzyHash (Rodríguez et al., 2018), which computes approximate matching hashes over Windows processes; PyREBox (Ugarte-Pedrero, 2017), which adds introspection to a virtual machine on top of QEMU using Volatility; Hollowfind (A., 2016), which detects different types of process hollowing techniques; or ShimCacheMem (House et al., 2015), which parses the Windows Application Compatibility Database from memory dumps.

The Windows Registry

Windows OS relies on the *Windows Registry* from Windows 3.1 onward. The *Windows Registry* is a hierarchical database that contains critical data for the normal operation of Windows and other applications, storing data regarding system booting, system configuration, and (per-user) software configuration (Russinovich et al., 2012).

Internally, this database is divided into files called *hives*, which contain a registry subtree. During system start-up, Windows maps these hives – stored on disk – into the system memory as a tree-like structure (in particular, as binary trees). These trees are further populated and assembled in the memory during system start-up, thus making up the complete view of the Windows Registry (Russinovich et al., 2012). Therefore, the Windows Registry is an in-memory representation of the on-disk hives with little differences which increase as more registry keys and values are added during system run-time. Furthermore, some of the registry keys can be *volatile keys*, that is, their information is stored only in the memory and is not preserved on disk when the corresponding registry hive is unloaded.

The root of each binary tree matches with one of the six predefined root keys of the Windows Registry:

- Hive `HKEY_LOCAL_MACHINE` (HKLM): this volatile hive contains the configuration information settings related to the local computer (i.e., software installed), as well as for the Windows OS itself. This hive is divided into different subkeys, according to the information that they store: security, Windows system setup, or software and Windows-user settings, among other things.

- Hive HKEY_CLASSES_ROOT (HKCR): this hive contains the associations of file name extensions and COM class registration information, indicating the applications used to handle these items. This registry key exists primarily for backward compatibility with 16-bit Windows and merges the information from HKEY_LOCAL_MACHINE\Software\Classes (default settings) with the information from HKEY_CURRENT_USER\Software\Classes (current user settings) (Microsoft Docs, 2018c). Depending on the execution privileges of the application and the status of the User Account Control, the COM run-time may ignore the per-user COM configuration and access only the per-machine COM configuration.
- Hive HKEY_CURRENT_CONFIG (HKCC): this hive stores information about the hardware profile currently being used, and mainly exists for backward compatibility. In fact, it is a pointer to the HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Hardware Profiles\Current registry key, providing quick access to it.
- Hive HKEY_PERFORMANCE_DATA (HKPD): this registry root key is related to performance counters. In particular, it contains runtime and performance data provided by the Windows kernel, drivers, apps, and services. All this information is volatile.
- Hive HKEY_USERS (HKU): this hive contains a set of subkeys, each of which corresponds to each of the user profile actively loaded in the machine. The subkey names match the security identifiers of every user. The settings and options for the currently-loaded user profile matches the currently signed-in user, thus restricting access to other user settings.
- Hive HKEY_CURRENT_USER (HKCU): this hive stores configuration information specific to the currently signed-in user's profile, such as the disk locations for user folders, control panel settings, and specific application configuration settings (Halsey and Bettany, 2015). In particular, this hive indicates the HKU subkey of the currently signed-in user. Roughly speaking, it can be seen as a symbolic link to that subkey.

The Windows Registry has two configuration scopes, system and per-user. *System hives* are usually stored in %SystemRoot%\System32\Config, while user profile hives (Ntuser.dat file) are stored under the %USERPROFILE% file directory. When a new user account is created in the system, a file named Ntuser.dat is also created and set up to represent his/her specific user configuration. This user configuration is later loaded during the system start-up process. In the event of having two different configurations for the same element, the system configuration prevails upon the user configuration.

The mapping process from on-disk hives into memory is part of the Windows boot process, and occurs at four different times (Russinovich et al., 2012): (1) during the early boot process stage, to load boot drivers before loading the Windows kernel. Here, the Winload process, which conducts the first part of the Windows boot process, reads the on-disk HKLM\SYSTEM hive to determine which device drivers need to be loaded to accomplish the boot and maps that hive into the memory; (2) during the Windows kernel boot stage, to load all drivers and tune the system according to its configuration. In this step, the Windows kernel populates the in-memory HKLM\HARDWARE hive with the initial hardware tree data passed by Winload; (3) during the user sign-in, to load the specific user configuration. This step is performed by the configuration manager, which fleshes out the in-memory Windows registry to include all its keys (in particular, loading on-disk HKLM\SAM, HKLM\SECURITY, and HKLM\SOFTWARE hives); and (4) during programs start-up, to know how the system is configured and then behave accordingly. Here, the interactive Windows logon manager process (named Winlogon) reads the in-file hive from the user profile currently signed in and maps it into HKCU.

Volatility and the Windows Registry

Volatility provides a set of functions to work with the in-memory representation of the Windows Registry. These functions allow a developer to obtain the current system configuration, return a registry key, enumerate registry subkeys, return a specific value of a registry key, enumerate keys showing also their last written date, and retrieve all keys in last modified order.

The current system configuration is retrieved through the CurrentControlSet registry tree. It contains the current system configuration such as device drivers, services, and kernel-/user-mode subsystem configuration settings. This registry tree is in fact a symbolic link to one of the configuration copies maintained by the Windows registry, located at the HKLM\SYSTEM\ControlSet00X registry path (being X 1 or 2). Usually, ControlSet001 contains the control set used in the last system booting, whereas ControlSet002 contains the control set used in the last system booting that was successful. Hence, if changes made to CurrentControlSet end up with a Windows OS unable to boot properly, the last configuration that allowed Windows OS to boot appropriately is still available in ControlSet002, and thus it allows Windows to roll back to a working state.

As explained before, there are some on-disk hives which are mapped into the memory during Windows start-up. When working with Volatility and the Windows Registry, we need first to specify the on-disk hive to analyze. For instance, we can use SOFTWARE or SYSTEM as values to access to the in-memory HKLM\SOFTWARE or HKLM\SYSTEM hives, respectively. Similarly, we can specify Ntuser.dat and a user name to analyze the per-user configuration (that is, to retrieve his/her particular HKU subkey). Recall that the name Ntuser.dat is the on-disk user profile hive, stored in the % USERPROFILE% file directory. When no user name is provided, the all-users configuration files are analyzed (i.e., the HKU root key is retrieved).

A taxonomy of Windows auto-start extensibility points

In this section we provide a taxonomy of Windows ASEPs, as well as a detailed explanation of each one. Recall that an auto-start extensibility point is a place which allows a particular program to run automatically in the system without any explicit user interaction. For instance, programs such as system services or scheduled tasks are transparently executed after system start-up, user logging, or the occurrence of certain trigger events.

Windows ASEPs can be classified according to the specific OS features used or abused by malicious programs to persist in the system. We have considered four categories: *system persistence mechanisms*, *program loader abuse*, *application abuse*, and *system behavior abuse*. For each category, we have distinguished different extensibility points.

Some extensibility points are only writable by programs with enough privileges. Furthermore, there are extensibility points that allow persistent programs to execute with elevated privileges, while in others the permissions of the signed-in user are inherited. The taxonomy that we propose is independent of the type (disk or memory) of forensics analysis. All the Windows ASEPs considered here are tracked down in a disk forensics analysis. However, some of these are undetected in the memory forensics of the Windows Registry (that is, we cannot tell which is the program being launched by that ASEP). Besides, it may happen that the ASEPs indicated as detectable in memory forensics could eventually be mapped out of the memory due to memory paging issues, and thus be undetected. As shown in the seminal work on memory forensics

and the Windows Registry by Dolan-Gavitt (2008), a large number of applications running in a system may cause unused portions of the registry to be paged out to disk.

Some extensibility points can require *the freshness of the system*, i.e., the system has to be rebooted or the user session has to be signed out and then signed in again to launch the programs set in these extensibility points. We indicate the minimum required freshness of the system for each technique (i.e., if an extensibility point requires only the user session to be restarted, it is obvious that a system reboot would have the same effect). The execution scope of an extensibility point is system-wide if any program in the system can interact with the program defined in that extensibility point. Otherwise, the execution scope is application-wide. Finally, certain extensibility points have different configuration scope, since they can be configured at system-level or at user-level, that is, they affect all the system as a whole or the current (signed-in) user session, respectively.

Table 1 summarizes the taxonomy of Windows ASEPs presented in this paper and their characteristics. In the rest of this section, we describe each category and explain the extensibility points that we consider, detailing also their characteristics.

System persistence mechanisms

This category describes the well-known mechanisms provided by Windows to run user programs, privileged tasks, or system services, among others programs. In this category we have included the following extensibility points: *run keys*, *start-up folders*, *scheduled tasks*, and *services*. We explain them in detail in the following subsections.

Run keys

The *run keys* are (almost certainly) the most well-known extensibility point. These registry keys include the `Run`, `RunOnce`, and `RunOnceEx`, which cause programs to run every time that a new user signs in the system (Microsoft Docs, 2018e). Unlike the `Run` registry values, the `RunOnce` registry values are deleted by Windows once the program starts (or optionally, when it finishes) its execution. Hence, malware using the `RunOnce` registry key has two options to persist in the next system start-up: either to write the registry value again during the program execution or to return a failure upon finishing execution (this would work if the deletion of the registry value was deferred until after the program execution). Lastly, `RunOnceEx` clears its registry values on completion of the program and does not create a separate process for each program defined in the registry key.

These run keys are located in the `HKCU` and the `HKLM` root registry keys (that is, they have different configuration scopes), and particularly in the `Software\Microsoft\Windows\CurrentVersion` (`Run` or `RunOnce`) registry paths.

Unlike the registry keys of the `HKCU` hive (user configuration scope), those created in the `HKLM` hive need admin privileges (system configuration scope). The execution privileges of the programs are equal to the privileges of the signed-in user. The content of these registry keys may be present in the memory, so they could be detected in memory forensics. The programs under these registry keys require the user to restart his/her session in order to execute, since they are launched every time that the user signs in the system. The execution scope of this extensibility point is limited to the application itself.

Startup folder

This extensibility point refers to a special Windows folder in which every program or application shortcut contained in the folder is launched during system start-up.

Like *Run keys*, this extensibility point has two different

Table 1
A taxonomy of Windows ASEPs and a summary of their characteristics.

	Characteristics					
Windows Auto-Start Extensibility Points	**Write permissions**	**Execution privileges**	**Tracked down in memory forensics**[†]	**Freshness of system**	**Execution scope**	**Configuration scope**
System persistence mechanisms						
Run keys (HKLM root key)	yes	user	yes	user session	application	system
Run keys (HKCU root key)	no	user	yes	user session	application	user
Startup folder (%ALLUSERSPROFILE%)	yes	user	no	user session	application	system
Startup folder (%APPDATA%)	no	user	no	user session	application	user
Scheduled tasks	yes	any	no	not needed[‡]	application	system
Services	yes	system	yes	not needed[‡]	application	system
Program loader abuse						
Image File Execution Options	yes	user	yes	not needed	application	system
Extension hijacking (HKLM root key)	yes	user	yes	not needed	application	system
Extension hijacking (HKCU root key)	no	user	yes	not needed	application	user
Shortcut manipulation	no	user	no	not needed	application	user
COM hijacking (HKLM root key)	yes	any	yes	not needed	system	system
COM hijacking (HKCU root key)	no	user	yes	not needed	system	user
Shim databases	yes	any	yes	not needed	application	system
Application abuse						
Trojanized system binaries	yes	any	no	not needed	system	system
Office add-ins	yes	user	yes	not needed	application	user
Browser helper objects	yes	user	yes	not needed	application	system
System behavior abuse						
Winlogon	yes	user	yes	user session	application	system
DLL hijacking	yes	any	no	not needed	system	system
AppInit DLLs	yes	any	yes	not needed	system	system
Active setup (HKML root key)	yes	user	yes	user session	application	system
Active setup (HKCU root key)	no	user	yes	user session	application	application

[†] *If the memory is paging to disk, it would be not possible to track down these ASEPs in memory forensics.*
[‡] *Depends on the trigger conditions defined to launch the program.*

configuration scopes. Each scope has a different root folder, but the same subpath is explored, namely `Microsoft\Windows\Start Menu\Programs\Startup`. The programs or application shortcuts in the path having `%APPDATA%` as a root folder are executed when the user signs in the system, while those having `%ALLUSERSPROFILE%` as a root folder are executed for all signed-in users. As before, the former folder (user configuration scope) does not require any special privileges, while the latter (system configuration scope) requires admin privileges to be written.

The execution privileges of these programs are equal to the privileges of the signed-in user. Since this extensibility point refers to a folder, the programs (or application shortcuts) contained in the folder are only known during a disk forensics analysis. However, note that in the case of a running program captured in the memory dump, we may obtain its execution path and check if it matches with the startup folder path (at the moment, our tool does not perform these checks). Like the *Run keys* extensibility point, this persistent mechanism needs the user session to have been signed out and then signed in again, and its execution scope is limited to the application itself.

Scheduled tasks

In Windows, a scheduled task is a program that executes periodically when certain conditions are met (known as *trigger conditions*). These tasks are executed by the Windows Task Scheduler (Microsoft Docs, 2018g), which is responsible for monitoring the system to observe when the criteria chosen to initiate the tasks are met. Different trigger conditions can be defined, such as launching a task at specific events, at a specific time with different regularity (daily, weekly, monthly, or monthly day-of-week schedule), when the system is booted, or when a user signs in.

For every scheduled task in the system, an XML file is placed in a folder inside the path `%SystemRoot%\System32\Tasks`. The XML file describes the program to execute (with arguments), the specific trigger conditions, and the security context that will be provided for that task.

Admin privileges are required to register a new scheduled task into the system. Regarding the execution privileges, a scheduled task runs at the privilege defined by the task security context. Recall that the specific details of a scheduled task are stored in an XML file on the disk. In the Windows Registry, however, only the scheduled task names and checksum information are stored. Therefore, we cannot know the scheduled tasks of a system through memory forensics. Unlike the previous extensibility points, this ASEP does not require any freshness of the system (unless system booting or user logging are set as conditions). Regarding the execution and configuration scopes, its execution is limited to the application itself, while it affects all the system.

Services

Windows services are background programs that have no user interaction. Furthermore, like scheduled tasks, they can be triggered when certain conditions are met. A service is usually devoted to a device driver software interface to communicate with specific hardware, but there may also be services not related to hardware communication. In Windows, any service must implement the interface functions provided by the Service Control Manager component, which is the Windows system process responsible for managing the start and the stop of services (Microsoft Docs, 2018f).

Services are executed with admin privileges which are also needed to register a new service in the system. The whole information regarding services is stored in the Windows Registry. Hence, memory forensics may retrieve all registered services from a memory dump. Like *scheduled tasks*, this extensibility point does not need freshness of the system (unless specified in the conditions). Its execution scope is limited to the application itself, while its configuration affects all the system.

Program loader abuse

These extensibility points rely on the techniques based on the abuse of the Windows program loader process. This process can be abused to transparently launch other programs when interacting with a user, without any notice to that user. We have included in this category the following ASEPs: *Image File Execution Options*, *extension hijacking*, *shortcuts manipulation*, *COM hijacking*, and *Shim databases*.

Image File Execution Options

The *Image File Execution Options* (IFEO) is a Windows feature that allows a developer to launch certain programs directly under a software debugger (Miskelly, 2005). The feature relies on a registry key located at `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options`. To enable it, a new registry subkey with the same name as the application to be debugged must be set in the registry path, as well as a registry value named Debugger having as data the full path to the software debugger. Hence, the software debugger is automatically launched when that application starts.

This feature can be abused since there is no verification as to whether the program set as a software debugger is indeed a debugger (Chen, 2005). For instance, if we want to execute the Windows calculator program automatically every time that the Mozilla Firefox browser is launched, we can set a registry subkey named `firefox.exe` and the registry value Debugger = "C:\Windows\System32\calc.exe".

The enabling of this Windows feature (i.e., writing the registry keys) needs admin privileges. The execution privileges are inherited from the user that launched the program to be debugged. Since this feature relies on the Windows Registry, it may be detected through memory forensics. This extensibility point does not need freshness of the system. Its execution scope is limited to the application itself, while its configuration affects all the system.

Extension hijacking

The *extension hijacking* attack consists in changing the default program associated with certain file extensions (for instance,`.txt`,`.docx`,`.pdf`, etc.) to another program. Hence, the program is automatically launched when the user opens these files. The associations of programs to file extensions are stored in the Windows Registry, specifically in the subkeys of `HKCU\Software\Classes\Applications` (user configuration scope) and in `HKLM\Software\Classes\Applications` (system configuration scope).

Unlike changes at user level, a change at system level requires admin privileges. The launched program always has the execution privileges of the user. As before, this extensibility point may be detected through memory forensics since it relies on the Windows Registry. This extensibility point does not require freshness of the system. Its execution scope is limited to the application itself.

Shortcuts manipulation

The *shortcuts manipulation* attack consists in manipulating the shortcuts that the user already has in the system (for instance, in the Desktop or in the quick launch taskbar), changing the target application by adding another program. To do so, the original application and the new program can be concatenated with the '; ' character. When Windows interprets the command, both programs are launched in parallel.

This ASEP has a user configuration scope and thus admin

privileges are not needed to modify the shortcuts of the user. However, this extensibility point can only be tracked down by a disk forensics analysis. It does not need freshness of system. Its execution scope is also limited to the application itself.

COM hijacking

The Microsoft Component Object Model (COM) is a subsystem that allows a developer to create software components (such as dynamic link libraries, DLLs for short) that can interact with others (Microsoft Docs, 2018h). COM software is implemented as a server/client framework, where the server process creates a COM server object to communicate with clients, who implement an equivalent COM client object. COM objects have a unique identification called *class identifiers* (CLSIDs), which represent a unique system resource (for instance, the Windows *My Computer* component has the CLSID value `20d04fe0-3aea-1069-a2d8-08002b30309d`).

The COM identifiers are stored in two different root keys in the Windows Registry, depending on the configuration scope: machine-wide COM objects are stored in `HKLM\Software\Classes\CLSID`, while per-user COM objects are registered in `HKCU\Software\Classes\CLSID`. Other CLSIDs may also exist in the `HKCR\CLSID` registry key due to backward compatibility issues. Moreover, each CLSID has an associated subkey in these registry paths named `InProcServer32` which defines the DLL implementing the COM server interface. The COM hijacking consists in modifying the path of the associated DLL to other malicious DLLs. Hence, when a program interacts with the hijacked COM object, the malicious DLL is loaded in the address space of the program.

Recall that the registry keys of the HKCU hive (user configuration scope) do not require privileges to be modified, unlike those of the HKLM hive (system configuration scope). Execution privileges are inherited from the user (per-user COM objects) or from the application that loads the hijacked machine-wide COM object. This extensibility point can be tracked down by memory forensics and does not need any freshness of the system to take place. Note that the execution scope is the system, since the hijacked COM object can interact with any other program.

Shim databases

Shim databases are designed to apply patches to a specific binary program prior to its execution (Microsoft Docs, 2012; McWhirt et al., 2017). Prior to the program execution, the Windows loader checks if there is any patch required for the program. If so, it applies the patch before executing the program. Shim databases are stored on disk in a path having as root folder `C:\Windows\AppPatch`. The subfolder that contains the databases depends on the underlying architecture, `Custom` (32-bit) or `Custom64` (64-bit).

There are two ways to register a new shim database: using the *sdbinst.exe* Windows utility or directly manipulating the Windows Registry. The latter involves artificially recreating the work of the Windows utility creating two registry keys in the `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\` registry path. First, a new registry key must be created in the `InstalledSDB` subkey, containing a global unique identifier and a registry value with the name DatabasePath and the path to the specific database file as data. Another registry key must then be added in the `Custom` subkey with the name of the affected program, a registry value with the name ValueName and the database filename as data.

Recall that any registry modification of the `HKLM` hive requires admin permissions. Similarly, write permissions are needed to add files to the system root directory. The execution privileges are inherited from the patched program. Although the extensibility point can be seen in memory forensics, the patch code resides on disk and thus it cannot be tracked down. Freshness of the system is not needed, and the execution scope is limited to the patched application. In contrast, its configuration scope is the whole system.

Application abuse

Some popular legitimate programs have the capability to extend their features thanks to plugins extensions. This category describes the extensions of legitimate programs that are abused to persist in the system. In this category we have included the following ASEPs: *trojanized system binaries*, *Office add-ins*, and *browser helper objects*. We explain them in detail in the following subsections.

Trojanized system binaries

A trojanized system binary attack consists in modifying a legitimate, system binary program that is patched to load another unintended program. System DLLs are a common target of these attacks, since DLLs are automatically loaded into the memory address space of any process that has dependencies on them.

This attack usually modifies the legitimate DLL by inserting new assembly code in an empty zone and then changing the entry point of the DLL to the zone. The new assembly code usually saves the state of all registers, loads the malicious program (normally as a malicious DLL), and then restores the state of the registers to continue normal execution of the legitimate DLL.

To succeed in this attack, admin privileges are required since a system binary program is being modified. The execution privileges of the trojanized binary will be inherited from the program that loads such a trojanized binary. Since DLLs are loaded into the memory address space of processes that have dependencies on them, a trojanized DLL can be captured in a memory dump if it is running at the time of the memory snapshot. However, this ASEP cannot be directly tracked down in memory forensics. This extensibility point does not require any freshness of the system. Its execution and configuration scopes are the whole system, since any program in the system depending on the trojanized DLL would launch it on execution.

Office add-ins

The Office add-ins allow a user to extend the features of Microsoft Office applications and interact with content in Office documents. From Office (2013) onward, these add-ins are similar to a web application, that is, they consist of a static HTML and a piece of Office-compliant JavaScript code. These web components are run in the context of a browser in a sandbox (Microsoft Docs, 2018d).

In contrast, previous implementations of Office add-ins were based on Visual Basic, Visual Studio, and COM add-ins. These add-ins involve binary code that runs directly on top of the user's system. The binary code of these Office add-ins is usually in the form of a DLL file, stored in the `%ProgramFiles%\Microsoft Office\[OfficeVersion]\Addins\[ProgID]` file path. They act similarly to COM components: in addition to the COM registration (see Section 3.2), an Office add-in also needs to create a subkey in the `HKCU\Software\Microsoft\Office\[OfficeApp]\Addins\[ProgID]` registry path to register itself as an Office application. Moreover, it also needs to create another registry value to specify if the add-in is loaded on every Office start-up, when requested by the user, or only once on the next Office start-up.

The installation of Office add-ins needs admin privileges to be written into the `%ProgramFiles%` folder. The execution privileges are inherited from the Office application (that is, from the user who launched it). We can only retrieve the name of the COM component and its configuration during memory forensics. This extensibility point does not require freshness of the system. Its execution scope is limited to the Office application that loads the add-in, while it has a per-user configuration scope.

Browser helper objects

Browser Helper Objects (BHO) are DLL files that work as plugins for the Internet Explorer browser (up to Internet Explorer 11). A BHO needs to register itself as a COM server, setting its CLSID in `HKLM\SOFTWARE\Windows\CurrentVersion\Explorer\Browser Helper Objects` (Microsoft Docs, 2007). At start-up, Internet Explorer loads all the registered BHOs after which the extensions can start to interact freely with the browser.

Recall that admin privileges are needed to write the `HKLM` hive. As before, the execution privileges are inherited from the Internet Explorer application (that is, from the user who launched it). Since its installation relies on the Windows Registry, this extensibility point may be tracked down in memory forensics. It does not need any freshness of the system and its execution scope is limited to the Internet Explorer application itself, while its configuration scope is the whole system.

System behavior abuse

The extensibility points under this category take advantage of the Windows features to launch programs. In this category, we have included *Winlogon*, *DLL hijacking*, *AppInitsDLLs*, and *Active Setup*.

Winlogon

As explained in Section 2.2, Winlogon is the last process in reading the on-disk hives when loading the specific user settings. Apart from this, Winlogon can also be configured to launch certain programs every time that a user signs into the system. This behavior is configured in the `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon` registry path. Furthermore, a DLL can be implemented as a Winlogon notification package and thus receive event notifications from Winlogon directly, just setting registry values in the registry path appropriately (Microsoft Docs, 2018i).

Usually, the system program specified to be launched by Winlogon is `userinit.exe`. This is maintained in a registry key value (`Userinit`). Winlogon also provides another registry key value, `Shell`, in which by default the Windows desktop and file manager is set (`explorer.exe`).

Note that both registry values are editable by users with enough privileges and thus the programs launched by Winlogon can be modified. The programs inherit execution privileges from the signed-in user. Since this extensibility point relies on a registry key, it can be tracked down by memory forensics. In this case, it also needs a system reboot (or at least a user session sign out and then sign in again) to initiate. The execution scope is limited to the application itself, while the configuration scope is the whole system.

DLL hijacking

DLL hijacking attacks consist in abusing the DLL search order done by Windows (Microsoft Docs, 2018b). The search for resolving DLL dependencies follows this order by default: (i) first, Windows searches the DLL in the working directory (i.e., the directory from which the application was loaded); (ii) if not found, the system directory (normally, `C:\Windows\System32`) is checked; (iii) likewise, if not found the 16-bit system directory is inspected (normally, the `C:\Windows\System` folder); (iv) then, the Windows directory is reviewed (i.e., the environment variable `%WinDir%`); (v) if not found either, the current directory is examined; (vi) and lastly, the directories defined in the PATH environment variable are iterated and checked.

A legitimate DLL in any of these folders can be substituted with a malicious one. Thus, such a malicious DLL will be loaded every time that a program has any dependency on the substituted legitimate DLL.

This extensibility point may require write privileges, depending on where the legitimate DLL being substituted is located. Although memory forensics can detect hijacked DLLs (for instance, checking the path of every loaded module in every process captured in the memory dump), we consider this persistence technique as untracked since it is not based on any registry key. Execution privileges are inherited from the application which loads the hijacked DLL. Furthermore, it does not require any freshness of the system. Note that the execution scope and the configuration may also be the system, since the hijacked DLL can be loaded by any program within the system.

AppInit DLLs

AppInit DLLs is a Windows feature that allows any DLL to be loaded into the address space of every application with a user interface – that is, any application that loads `user32.dll`. Hence, this feature can be abused by a malicious DLL to ensure that it is loaded with every interactive application.

This feature resides in the `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs` registry key. Here there is a registry value `LoadAppInit_DLLs` that contains the full paths of the DLLs to be loaded. Let us remark that this feature is disabled from Windows 8 onward if the secure boot feature is enabled (Microsoft Docs, 2018a).

This extensibility point requires admin privileges, while execution permissions are inherited from the application which loads those DLL specified in the extensibility point. This ASEP may be detected in memory forensics, since it relies on the Windows Registry. Like the previous ASEP, it does not need any system boot or restart of the user session to take place. Regarding the execution and configuration scope, it affects the whole system in both cases.

Active Setup

Active Setup is another Windows feature that enables programs to be launched when a user signs in the system (Klein, 2010). This extensibility point can be configured at system or at per-user level, depending on the root registry key used (`HKLM` and `HKCU`, respectively) in the subpath `SOFTWARE\Microsoft\Active Setup\Installed Components`.

A new registry key must be created in these registry paths to enable this feature, having an Active Setup identifier (specifically, a global unique identifier) as a registry key. Two registry string-typed values must also be created, *Version* and *StubPath*. The data of the latter registry value indicates the program to be launched. Let us remark that in the case of having the same global unique identifier in both `HKLM` and `HKCU` hives, then the Version data values are compared. If that in `HKCU` is less than that in `HKLM`, then the program indicated in the `HKLM` hive is launched while the program indicated in the `HKCU` hive is ignored.

Recall that admin privileges are required to write to the `HKLM` hive, unlike the `HKCU` hive. The programs specified in this ASEP have the same privileges as the signed-in user. Like the previous ASEP, it may be detected in memory forensics since it relies on the registry. Regarding the freshness of the system, it needs at least the user session to be signed out and then signed in again. Its execution scope is limited to the application itself.

Experimental evaluation of `Winesap`

In this section, we first describe our tool `Winesap`. Then we evaluate the execution order of some registry-based Windows ASEPs and the effectiveness of our tool to detect Windows AESPs in memory dumps.

Description of `Winesap`

The tool `Winesap` is implemented as a Python plugin on top of Volatility, considering the particularities explained in Section 2.3. It is specially designed to track down the previously described Windows ASEPs that rely on the Windows Registry. Our tool has been released under the GNU GPLv3 license and is freely available at https://gitlab.unizar.es/rrodrigu/winesap.

The tool works as follows: first, it obtains all possible detectable registry-based Windows ASEPs in a memory dump. It then determines whether the detected extensibility points are being abused by malware. In this regard, the tool performs different checks, depending on the data type contained in the register key value under analysis.

A registry key value can store different data types, such as binary data (type `REG_BINARY`), numeric data (type `REG_DWORD` or `REG_QWORD`), or unknown encoded data (type `REG_NONE`), among others. When data is stored as binary or unknown, `Winesap` tries to parse the content data as a PE header. If it successes, it raises a warning message indicating that a suspicious program was detected. When data is stored as string data (types `REG_SZ`, `REG_EXPAND_SZ`, or `REG_LINK`), the tool assumes that the string refers to a path to an executable file. Hence, the tool analyzes the string and marks as suspicious those paths that contain any reference to the folders `%APPDATA%` (i.e., `C:\Users\[username]\AppData\Roaming`), `%TMP%` or `%TEMP%` (i.e., `C:\Users\[username]\AppData\Local\Temp`), or the common parent folder `AppData`.

Additionally, the tool also marks as suspicious any string command that contains operations regarding NTFS Alternate Data Streams (ADS) (Huebner et al., 2006) or well-known shell commands that indirectly launch programs (Mosuela, 2016; Reichel, 2017). For instance, the command `rundll32.exe shell32.dll,ShellExecute_RunDLL [filename]` will load the system library `shell32.dll` and will execute the function `ShellExecute_RunDLL` with the parameter filename. As a result, the filename starts to execute. This indirect execution is commonly used by malware and other post-exploitation APT agents (such as CobalStrike).

Experiments and discussion

As a test environment, we have used a VMware Workstation 14 hypervisor with a freshly installed Microsoft Windows 7 SP1 ×64 logged into a user account with administrator privileges. The use of a 64-bit virtualized system allows us to check the detection of ASEPs in both a 64-bit system and in a SysWow64 subsystem (32-bit programs running on top of a 64-bit system).

We have created a Python script for each of the Windows ASEPs described in this paper. These scripts make a copy of a binary file in a randomly created folder in the %APPDATA% path and then make sure that the binary persist in the system using a particular extensiblity point. As binary files for experimentation, we used `cmd.exe` or the system library `autoplay.dll`, depending on which kind of binary file was launched by the extensibility point. Furthermore, we executed those scripts with admin privileges to guarantee a successful installation in every extensibility point, regardless of its write permissions. These scripts are also provided in the repository of the tool for the sake of reproducible research.

As a first experiment, we used the scripts to populate every registry-based Windows ASEP in the virtual machine, aiming at empirically evaluating the execution order of the ASEPs when a user signs in the system for the first time. In this regard, every extensibility point has launched a test program designed to collect its unique process identifier (PID) and the creation timestamp. Our results indicate PIDs are sequentially assigned at Windows booting and the execution order obtained was as follows:

1. *Winlogon* (`Userinit` registry key value)
2. *Winlogon* (`Shell` registry key value)
3. *Run keys (HKLM root key;* `RunOnceEx` subkey)
4. *Run keys (HKCU root key;* `RunOnceEx` subkey)
5. *Run keys (HKLM root key;* `RunOnce` subkey)
6. *Active Setup (HKLM root key)*
7. *Active Setup (HKCU root key)*
8. *Startup folder (%ALLUSERSPROFILE%)*
9. *Startup folder (%APPDATA%)*
10. *Run keys (HKCU root key;* `Run` subkey)
11. *Run keys (HKLM root key;* `Run` subkey)
12. *Run keys (HKCU root key;* `RunOnce` subkey)

As a second experiment, we tested every Windows ASEP individually. The virtual machine was halted and its memory was retrieved after each script execution, obtaining a VMWare.`vmem` file. Then the memory dump was examined with `Winesap`. Finally, the virtual machine was reverted to a clean state to execute the next script.

Discussion of results and limitations

Our tool succeeded in retrieving all the Windows ASEPs introduced in this paper which are detected in memory forensics (that is, those that rely on the Windows Registry), regardless of their configuration scope. Each of these extensibility points was also marked as suspicious, since the programs indicated by the extensibility point matched some of the aforementioned suspiciousness conditions. An example of the output of the tool is as follows:

```
WARNING:
Suspicious path file
HKLM\Software\Microsoft\Windows NT\CurrentVersion\
Image File Execution Options\firefox.exe
Debugger: REG_SZ: C:\Users\me\AppData\Roaming
Yztrpxpt\cmd.exe
WARNING:
Suspicious path file
HKLM\Software\Wow6432Node\Microsoft
Windows\NT\CurrentVersion\Windows
AppInit_DLLs: REG_SZ: C:\Users\me\AppData
Roaming\Uxkgoeaoqbf\autoplay.dll
```

Note that some Windows ASEPs included in the taxonomy do not rely on the Windows Registry. Therefore, `Winesap` cannot check the original ASEPs which triggered the execution of some programs. However, we can still perform some actions to overcome this problem. For instance, Startup folder and Shortcut manipulation ASEPs are symbolic links pointing to other programs. These symbolic files are stored on disk, and hence only detectable in disk forensics. Nevertheless, if we perform a file carving in the memory dump, we may obtain resource files recently freed by the OS and check if those resources match with symbolic link files whose paths point to the startup folder or whose value has been manipulated.

Similarly, the scheduled tasks only store the task name and checksum data in the Windows Registry. Other data is defined in XML files, located in the system root folder. Hence, a file carving approach may also be useful in this case to obtain those XML files from the memory dump and then parse the information. Likewise, trojanized system binaries, DLL hijacking, and Office add-ins need a traditional memory forensics analysis focused on the running processes captured in the memory snapshot. Once a suspicious process is located, its current directory can be retrieved by means of Windows internal structures and then checked to see if it matches with some of these Windows ASEPs.

Related work

The term *Auto-Start Extensibility Points* was first introduced in (Wang et al., 2004) to describe the software mechanisms that allow programs to auto-start on Windows start up without explicit user invocation. In that work, four categories were itemized: (1) ASEPs that start new processes; (2) ASEPs that hook system processes; (3) ASEPs that load drivers; and (4) ASEPs that hook multiple processes. Besides, two types of ASEP hooking were also distinguished: as a standalone application that is automatically launched by Windows (usually due to registry keys in the Windows Registry); or an extension of existing programs that are either automatically run (for instance, by the process Winlogon) or commonly launched by users (for instance, Internet browsers). In this paper, we have introduced a more complete view of Windows ASEPs according to the system features used/abused by malware to persist in the system. Furthermore, we have also indicated the different characteristics of each of these extensibility points.

Various works in the literature have studied the methods used by malware to persist in a Windows system. For instance, the authors in (Kirda et al., 2006) studied the behavior of spyware that used the BHO extension of Internet Explorer to monitor the browsing behavior of a user. Spyware is a type of malicious software designed to gather data from a system and forward it to a third party, without the consent or knowledge of the user. Similarly, the works in (Sikorski and Honig, 2012; Russinovich and Margosis, 2016; Hasherezade, 2017; Monnappa, 2018) have summarized most of the Windows ASEPs found during malware analysis.

However, to the best of our knowledge, we are the first to propose a taxonomy of Windows ASEPs, focusing on their internal details and characteristics.

As regards ASEPs-related tools, most of the works described tools related to spyware. For instance, the authors in (Wang et al., 2004) introduced *Gatekeeper*, which monitors the ASEPs described in that work and complements the traditional signature-based approach to detecting spyware behavior. Similarly, the tool STARS was proposed in (Wu et al., 2007) to remove ASEPs used by spyware that is able to protect itlself against deletion.

A tool close to ours is Autoruns (Chopitea, 2014), another Volatility plugin aimed at finding out where malware persists. First, it analyzes the dump locating different ASEPs, and then it tries to match those ASEPs with the running process in the memory dump. Although our tool looks for more ASEPs, currently it does not cross check with processes as Autoruns does. We aim to adopt this good strategy in `Winesap` as future work.

Finally, it is worth mentioning the well-known tool `Autoruns for Windows` (Russinovich, 2017), from the Sysinternal's suite. This tool allows a user to check the full set of Windows AESPs in a live and running computer. Our tool `Winesap` works in a similar fashion, but analyzing a memory dump instead of a live system. Hence, our tool can help a memory forensics analyst during the triage phase to detect the presence of unknown and rare programs in those ASEPs, thus giving the analyst good insights into suspicious programs.

Conclusions

Memory forensics is the process of analyzing the memory dump of a system and searching for evidences after a computer security incident. Many such incidents are caused by malware, which can persist in the compromised system by different means. The techniques used to persist in the system and that allow a program to be launched without any explicit user invocation are known as auto-start extensibility points (ASEPs).

In this paper, we have reviewed the ASEPs provided by Windows and built a taxonomy including the OS features used or abused by malware. This taxonomy is independent of the type of forensics analysis (disk or memory). The taxonomy is divided into four categories: system persistence mechanisms, program loader abuse, application abuse, and system behavior abuse. For each category, we have distinguished different extensibility points. Each ASEP has been finally characterized according to its write permissions, execution privileges, detectability in memory forensics, freshness of system requirements, and execution and configuration scopes. Furthermore, we have developed a Volatility plugin named `Winesap` that analyzes a memory dump and highlights the registry-based Windows ASEPs with suspicious programs. Hence, this tool serves as a memory forensics triage tool, helping the memory forensics analyst to identify the programs located in the ASEPs contained in the memory dump. To foster research in this area and to help memory forensic analysts, `Winesap` is freely released under the GNU GPLv3 license.

As future work, we aim to improve the features of `Winesap` by means of process and file carving. We also aim to enable analysts to define the non-suspicious paths.

Acknowledgements

This work was supported in part by the University, Industry and Innovation Department of the Aragonese Government (*Programa de Proyectos Estratégicos de Grupos de Investigación*; DisCo research group, reference T21-17R).

References

A, M.K., 2016. Detecting deceptive process hollowing techniques using HollowFind Volatility plugin. Online. https://cysinfo.com/detecting-deceptive-hollowing-techniques/. (Accessed 23 October 2018).

AV-TEST Institute, 2018. Security Report 2017/2018. techreport. Available at: https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2017-2018.pdf. (Accessed 23 October 2018).

Chen, R., 2005. Beware the Image File Execution Options key. Online. https://blogs.msdn.microsoft.com/oldnewthing/20051219-11/?p=32923. (Accessed 23 October 2018).

Chopitea, T., 2014. Volatility Autoruns plugin. Online. http://tomchop.me/2014/09/18/volatility-autoruns. (Accessed 23 October 2018).

Cichonski, P., Millar, T., Grance, T., Scarfone, K., 2012. Computer Security Incident Handling Guide. techreport SP 800-61 Rev. 2. National Institute of Standards and Technology (NIST). Special Publication.

Dolan-Gavitt, B., 2008. Forensic analysis of the Windows registry in memory. Digit. Invest. 5, S26–S32.

Granc, T., Chevalier, S., Scarfone, K.K., Dang, H., 2006. Guide to Integrating Forensic Techniques into Incident Response. techreport 800-86. National Institute of Standards and Technology (NIST). Special Publication.

Halsey, M., Bettany, A., 2015. Windows Registry Troubleshooting, first ed. Apress.

Hasherezade, 2017. Wicked malware persistence methods. In: AppSec EU 2017.

House, F., Teodorescu, C., Davis, A., 2015. Shim Shady: Live Investigations of the Application Compatibility Cache. Online. https://www.fireeye.com/blog/threat-research/2015/10/shim_shady_live_inv.html. (Accessed 23 October 2018).

Huebner, E., Bem, D., Wee, C.K., 2006. Data hiding in the NTFS file system. Digit. Invest. 3 (4), 211–226.

Kirda, E., Kruegel, C., Banks, G., Vigna, G., Kemmerer, R., 2006. Behavior-based Spyware Detection. In: Proceedings of the 15th Conference on USENIX Security Symposium, vol. 15. USENIX Association, Berkeley, CA, USA, p. 16.

Klein, H., 2010. Active Setup Explained. Online. https://helgeklein.com/blog/2010/04/active-setup-explained. (Accessed 23 October 2018).

Ligh, M.H., Case, A., Levy, J., Walter, A., 2014. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. John Wiley & Sons, Inc.

McWhirt, M., Erickson, J., Palombo, D., 2017. To SDB, Or Not To SDB: FIN7 Leveraging Shim Databases for Persistence. Online. https://www.fireeye.com/blog/threat-research/2017/05/fin7-shim-databases-persistence.html. (Accessed 23 October 2018).

Microsoft Docs, 2007. The Basics of Browser Helper Objects. Online. https://blogs.msdn.microsoft.com/askie/2007/12/07/the-basics-of-browser-helper-objects/. (Accessed 23 October 2018).

Microsoft Docs, 2012. Understanding Shims. Online. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-7/dd837644(v=ws.10. (Accessed 27 October 2018).

Microsoft Docs, 2018a. AppInit DLLs and Secure Boot. Online. https://docs.microsoft.

com/en-us/windows/desktop/dlls/secure-boot-and-appinit-dlls. (Accessed 23 October 2018).
Microsoft Docs, 2018b. Dynamic-Link Library Search Order. Online. https://docs.microsoft.com/en-us/windows/desktop/dlls/dynamic-link-library-search-order. (Accessed 23 October 2018).
Microsoft Docs, 2018c. Merged View of HKEY_CLASSES_ROOT. Online. https://docs.microsoft.com/en-us/windows/desktop/sysinfo/merged-view-of-hkey-classes-root. (Accessed 23 October 2018).
Microsoft Docs, 2018d. Office Add-ins platform overview. Online. https://docs.microsoft.com/en-us/office/dev/add-ins/overview/office-add-ins. (Accessed 27 October 2018).
Microsoft Docs, 2018e. Run and RunOnce Registry Keys. Online. https://docs.microsoft.com/en-us/windows/desktop/setupapi/run-and-runonce-registry-keys. (Accessed 23 October 2018).
Microsoft Docs, 2018f. Services. Online. https://docs.microsoft.com/en-us/windows/desktop/services/services. (Accessed 23 October 2018).
Microsoft Docs, 2018g. TaskScheduler. Online. https://docs.microsoft.com/en-us/windows/desktop/taskschd/task-scheduler-start-page. (Accessed 23 October 2018).
Microsoft Docs, 2018h. The Component Object Model. Online. https://docs.microsoft.com/en-us/windows/desktop/com/the-component-object-model. (Accessed 23 October 2018).
Microsoft Docs, 2018i. Winlogon Notification Packages. Online. https://docs.microsoft.com/en-us/windows/desktop/secauthn/winlogon-notification-packages. (Accessed 23 October 2018).
Miskelly, G., 2005. Inside 'Image File Execution Options' debugging. Online. https://blogs.msdn.microsoft.com/greggm/2005/02/21/inside-image-file-execution-options-debugging. (Accessed 23 October 2018).
Monnappa, K.A., 2018. Learning Malware Analysis: Explore the concepts, tools, and techniques to analyze and investigate Windows Malware. Packt Publishing.
Mosuela, L., 2016. How It Works: Steganography Hides Malware in Image Files. Online. https://www.virusbulletin.com/virusbulletin/2016/04/how-it-works-steganography-hides-malware-image-files. (Accessed 23 October 2018).
Reichel, D., 2017. 2016 Updates to Shifu Banking Trojan. Online. https://researchcenter.paloaltonetworks.com/2017/01/unit42-2016-updates-shifu-banking-trojan. (Accessed 23 October 2018).
Rodríguez, R.J., Martín-Pérez, M., Abadía, I., 2018. A Tool to Compute Approximation Matching between Windows Processes. In: Proceedings of the 2018 6th International Symposium on Digital Forensic and Security. ISDFS), pp. 313–318.
Russinovich, M., 2017. Autoruns for Windows v13.91. Online. https://docs.microsoft.com/en-us/sysinternals/downloads/autoruns. (Accessed 23 October 2018).
Russinovich, M., Margosis, A., 2016. Troubleshooting with the Windows Sysinternals Tools. Microsoft Press.
Russinovich, M.E., Solomon, D.A., Ionescu, A., 2012. Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7, sixth ed. Microsoft Press, Redmond, WA, USA.
Sikorski, M., Honig, A., 2012. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software, first ed. No Starch Press, San Francisco, CA, USA.
Sood, A.K., Bansal, R., Enbody, R.J., 2013. Cybercrime: Dissecting the State of Underground Enterprise. IEEE Internet Computing 17 (1), 60–68.
Ugarte-Pedrero, X., 2017. PyREBox, a Python Scriptable Reverse Engineering Sandbox. Online. https://blog.talosintelligence.com/2017/07/pyrebox.html?m=1. (Accessed 23 October 2018).
Walters, A., Petroni, N., 2007. Volatools: Integrating Volatile Memory Forensics into the Digital Investigation Process. In: BlackHat DC.
Wang, Y.M., Roussev, R., Verbowski, C., Johnson, A., Wu, M.W., Huang, Y., Kuo, S.Y., 2004. Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management. In: 18th USENIX Conference on System Administration. USENIX Association, Berkeley, CA, USA, pp. 33–46.
Wu, M.-W., Wang, Y.-M., Kuo, S.-Y., Huang, Y., 2007. Self-Healing Spyware: Detection, and Remediation. IEEE Trans. Reliab. 56 (4), 588–596.