Longitudinal Study of Persistence Vectors (PVs) in Windows Malware: Evolution, Complexity, and Stealthiness

Nicholas Phillips
Department of Computer and Information Sciences
Towson University
nphill5@students.towson.edu

Aisha Ali-Gombe
Division of Computer Science and Engineering
Louisiana State University
aaligombe@lsu.edu

***Abstract*—Malware is the driving force for most cyber-attacks and, in recent years, has continued to be one of the most challenging threats facing our cyber infrastructure. Modern malware's adaptive design often leverages complex and evolving technologies to overcome various detection and preventive security tools. One of these techniques is Persistence - an ability to survive on victim systems past the current power cycle. The persistence vector allows the malware to live on host machines without detection. Thus, this paper conducts a longitudinal study and characterization of Windows malware Persistence Vectors (PVs) across more than 1000 malware samples. We explored the evolution, complexity, and stealthiness of persistence vectors in modern Windows malware families using the combination of static and dynamic analysis. The result of our study indicated that security tools and analysts could utilize PVs as decoys to strengthen malware defensive strategies.**

Keywords: Malware, Persistence Vectors, System Security, Reverse Engineering

I. Introduction

Malware is an ever-evolving threat against cyber infrastructure. With nearly one billion malware attacks in 2021, and predictions show that, with the rise in remote work, this number is forecasted to increase a minimum of ten percent over the next year, making the ever-growing threat more daunting [7]. Current defensive measures are predominately positioned at the perimeter of networks and scanning the system attempting to stop potential malware infections [1]. However, they have an extensive blind spot in dealing with malware once it obtains a foothold on the system. New generation malware, especially the Rootkit class, leverages variable stealth and mutation strategies to persist after infection. With these advantages, coupled with vulnerabilities present on the system and those introduced via users, security is constantly on the back foot in the endless cycle of attack and defense. Therefore, the practice of identifying, extracting, and utilizing the persistence mechanisms in defensive measures is one massive step towards leveling the field.

The rest of the paper is organized as follows: Section 2 presents the problem statement; Section 3 provides an analysis of the current research into malware; Sections 4 presents a delve into the background of persistence vectors in malware; Section 5 and 6 presents our data collection and analysis of persistence vectors, respectively; and Section 7 presents future works and concludes the paper.

II. Problem Statement

As security works to develop methodologies to stop malicious threats from obtaining access, malware authors deploy new methods, such as those presented via zero days [4] [5] [6] [13] . In 2021, record numbers of zero days utilized, 64 confirmed, and untold number unconfirmed [31]. This cycle resets with malicious actors constantly holding the edge by only needing one compromise to be victorious. Even major advancements, such as Secure Boot have proven insufficient and susceptible to compromise. Theoretical and wild bootkits have generated means around this improvement, such as forged certificates or enabling their loading prior to the safe image Security Boot loads [16]. Attention primarily focused on the exterior surface with attempts to stop malware from infecting the system. Only a small amount of focus has been paid to internal areas where the attacks land. Persistence vectors, while not deeply diverse as developed attack vectors, have undergone a constant evolution, and remained unanalyzed. Thus, we present a longitudinal analysis on the evolution of Windows malware persistence vectors providing new insight into their complexity and stealthiness. The objective of our study is to provide a new direction for malware defensive capabilities leveraging persistence vectors rather than the traditional payload and infection vector scanning.

III. Related Work

Literature dealing with malware persistence is limited in content, which is presented below. Gittins and Soltys conducted one of the few pieces of research into malware persistence mechanisms. They analyzed the more common currently used malware persistence elements, and some are believed to be utilized by Nation State actors through a showing of independent samples for each of the presented persistence mechanisms [2]. While the illustrated persistence vectors are accurate, the sample base shown is only five samples deep, leaving it only as an overview, not in-depth. Rana et al. presented research into persistence mechanisms in conjunction with obfuscation techniques. Based on the solar wind attack, they cover various persistence utilized on Windows systems, with proposed solutions to attempt to minimize the effects of persistence vectors identified [3]. While the persistence vectors covered are extensive and the suggested solutions can help mitigate malware persistence, there is a shortcoming in that malware continues to evolve and tend to avoid detection using different obfuscation techniques. Khushali presented research into the subsection of malware titled fileless malware. This malware often does not write to the persistent storage, making them harder to detect [29]. Although very stealthy, fileless malware remain present on a victim system until the next power cycle. While these types of malware are useful for small campaigns, persistence is still vital for more prolonged operations generally utilized by nation-state actors and more extensive malware campaigns. Kohout and Pevný used persistence implanted web traffic as means of identifying long-placed malware [30]. While this does provide an effective means of identifying infected systems, it is limited to targeting the web traffic and not dealing with the various other persistence mechanisms. Our study presents a deeper analysis, utilizing a more comprehensive sample base than previously used and including means of relating persistence to stealth measures as well.

The remaining literature referencing persistence is focused on two main categories of research: (1) Research into malware's functionality, such as anti-analysis techniques, API modifications, or evasion techniques, and (2) Deeper analysis into a specific malware family/sample. Maffia [24], Mills [23], and Galloro [22] researched into the evolution of malware evasion techniques over the years. Mills developed a sandbox modification tool titled MORRIGU, which is utilized to subvert the malware evasion techniques, specifically those that prevent malware from executing in an analysis environment [23]. Analysis tools such as this, and some of its predecessors such as HookMe, Cuckoo Sandbox, and PyREBox, are excellent at dealing with defensive measures that malware deploys to prevent its analysis [25]. However, they are designed with extensive implementation and configuration changes, making them difficult to configure. These analysis environments are also designed to detect malware behavior mostly from a payload standpoint. However, they quickly become obsolete because modern malware evolves and employs sophisticated obfuscation techniques. Galloro et al. study the history and development of various evasion techniques. By completing the analysis comparison, they produced listings of evasion techniques only utilized via malware [22]. Maffia also conducted research along similar lines. The authors proposed PEPPER - a Pintool designed to defeat standard malware evasion techniques, such as Anti-VM [24]. Both provide excellent detailing of evasion techniques; however, as with the analysis environments, they detect from the payload standpoint. These analysis tools may provide false negatives if the evasion techniques have changed.

IV. Background on Persistence Vectors

Persistence vectors are sections of code built within software packages (both legitimate and malicious) that allow programs to survive system restart, switching between users, and similar system start-up functionality. In general, persistence is achieved by modifying certain sections of the system or kernel data structure. This section will enumerate and discuss the most commonly used persistence vectors. The complete listing of known Windows persistence vectors can be found in the MITRE ATT&CK framework® [15].

A. Common Persistence Vectors

1) Registry modification : Registry modifications are the most common persistence mechanisms utilized by malicious code [28]. By adding a value to a specific registry key, malicious code can ensure either it is loaded upon start, it is utilized before legitimate files, or it is reinstalled after being deleted. An example of this is the entry of a modification in the *run* key. These values can be set under *HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\* or *HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\* for the following keys:

- *HKCU\....\Run*
- *HKCU\....\RunOnce*
- *HKLM\....\CurrentVersion \Run*
- *HKLM\....\RunOnce*
- *HKLM\.... \PoliciesExplorer\Run*

2) DLL Replacement/Reorder : The next common persistence method is Dynamic Link Library (DLL) Hijacking. This vector works with modification or complete replacement of vulnerable DLLs with malicious code. When the modified DLL is called the malicious code is loaded and executed. A secondary method utilizing DLLs is through the DLL search order hijacking, where the original DLL remains intact but is dropped in priority for the malicious version placed on the system.

3) Startup Keys : Start-up key and service modification vectors utilize a combination of the above two techniques by setting the malicious code into a priority slot in boot order. Once loaded the malicious code is restarted on the system, maintaining the infection.

Files under the startup directory can have a shortcut created to the location pointed by subkey of startup. If this value is present then the service will launch during a system reboot. These values can be set under *HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\* or *HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\* for the following keys:

- *HKCU\....\Explorer\ShellFolders*
- *HKCU\....\Explorer\User\ShellFolders*
- *HKLM\....\Explorer\ShellFolders*
- *HKLM\....\Explorer\User\ShellFolders*

B. Services

Several Windows services are required to be started at boot for the system to run properly. By placing any malware keys in this startup folder it is able to execute at startup as other services. Additionally, because alternative services can be started if another fails to load, a malware author can append these failed states to launch the malware.

1) Boot Modifications : Bootkits and other malware have begun utilizing a type of alteration called boot key modifications. In persistence technique, the smss.exe launches before the Windows subsystem, calling the configuration subsystem to load the hive present at *HKLM\SYSTEM \CurrentControlSet\Controlhivelist*. Any value that contains the *BootExecute* key will be launched at system boot by the smss.exe via the *HKLM\ControlSet002\ControlSession Manager* [27]. A normal system should only have the value of autocheck or autochk*.

2) Shortcut Creation/Modification : Shortcut hijacking obtains persistence via rewriting of saved icons of applications that users commonly use. This is created through either replacing the direct calling program with a compromised version or a wholly malicious one.

3) Event Trigger Execution : One of the oldest forms of persistence is the event-triggered execution technique. This method achieves persistence on a system via the setting of time-trigger automation, such as CHRON to launch upon the system restart or through program infection when another program is launched, a redirect to the malware code is present, restarting it.

4) Kernel Module Changes : Although much more challenging to implement, malware can leverage changes in the kernel module to achieve persistence. In this technique, the malware hides its presence by loading modules from the default order to include the malware as a high-value loaded item, either as a change to the BIOS load order or through appending BOOTSTRAP code. While this is often not as pervasive as the other techniques discussed due to the possibility of a system crash, it is nonetheless one of the most effective persistence vectors.

V. Data Collection and Analysis

As stated in the introduction, this paper aims to systematize knowledge for Windows malware persistence vectors in a longitudinal study. We will analyze the persistence vector's characterization, complexity, and stealthiness. The overarching goal is to drive new knowledge in understanding the metamorphosis of persistence vectors that will help design new malware defensive strategies. Thus, for the data collection, we downloaded a total of one thousand malicious software from virus repositories VirusShare and VirusTotal [26]. All are samples from within the past ten years, yielding a solid base for the evolution of malware over time, and were selected to run the spectrum of malware families. Each sample was manually processed via static and dynamic malware analysis to extract APIs, data, and metadata. Then the sample is passed to IDA Pro for the actual persistence vector extraction. Finally, a detailed code reconstitution is performed.

1) Environment Setup: The PV extraction process is carried out on Windows 7 and 10 Virtual Machines (VMs), with two copies of each: one for dynamic analysis and one for static analysis. Each machine has two 2.4 GHz cores and 4 GB RAM. Additional steps were taken using Hidetoolz to minimize the effect of Anti-VM and Anti-Reversing during the analysis [10]. Configuration settings were then modified through the utilization of the Vbox info modifier capability. This allowed for the default options, such as the system utilized and system manufacturer searched for via VM aware malware, to be changed. Additional VM capabilities, such as the Addons, were removed and the sub-keys in the registry deleted. Commonly installed user software were added, attempting to give the appearance of a real system instead of a virtual one.

2) Static Analysis: Static analysis is defined by collecting information about the binary, precisely a malware sample in this case, without executing or creating a runtime memory space [12]. Before analysis, the first step is to collect a file hash in the form of MD5 and/or SHA256. This step ensures that the file downloaded matches the one presented by the collection sites of VirusTotal and Malshare. We utilized the Powershell command - *get-filehash* to accomplish the hashing process. Next, for each target malware, we ran it against an unpacker to remove any possible common packers and cryptors, leaving behind the bare-bones malware code that the analysis tools would evaluate. For this we utilized PEId to identify and remove the common packers and compression utilized by the malware samples. In the static analysis of the Rovnix bootkit for instance, we found the hash to be 7CFC801458D64EF92E210A41B97993B0, and PEID identified that two packers were used in the initial sample.

Immediately after unpacking, a target sample is then executed against the *Strings* utility. This utility allows for ASCII and Unicode string identification. In this task, we are looking for specific Windows API calls which can be tied back to the potential persistence modifications and additional files created, which could contain remaining malicious payloads. The sample is also loaded into Dependencies, a modern rewriting of Dependency Walker, which identifies utilized DLLs for the executable. This tool also determines the potential persistence vectors utilized via DLL replacements, and those utilizing the creation of files. In the Rovnix bootkit sample, the strings showed indications of file creation, such as the *CREATE* and *FILEACCESS APIs*, while Dependencies showed access to kernel-level modules, kernel32.dl and ntdll.dll. These match the boot modifications, driver deployment, and registry changes, along with the items needed to generate the malicious Volume Boot Record (VBR).

3) Dynamic Analysis: Dynamic analysis is completed by collecting binary elements while executing the file on the system. Before launching an executable, we first run a suite of malware analysis tools: ProcWatch, CaptureBatch, and RegShot. These tools create baseline analysis to compare modifications made by the malware sample in processes, batch files, and registry. After removing standard system processes, we can notice the unique ones created by the malware and registry modifications, if any. These creations are the specific items we targeted as the persistence functions of the malware as they show the specific files, DLLs, and registry keys that the malware implants to obtain persistence.

For Rovnix, we identified the file creations for the modified VBR and malicious DLLs. Additionally we found the registry modifications generating the boot changes, consisting of the backup copies of the malware code and the independent ones used to restart these backups if other elements were removed.

4) Persistence Identification and Extraction: The persistence vectors of the sample base are identified from this two tiered reverse engineering process. From here the process of removing these code segments is conducted. The samples are loaded in IDA Pro Disassembler, with the information gathered from analysis used to target the specific functions completing the persistence modifications. These functions are exported utilizing the inbuilt exporting capability in the HexRays loaded with IDA. Data like this can then be exported as raw text or as set variables or segments of C code. These individual identified persistence vectors are saved with the naming convention of ”persistence vector-file hash”. Each PV was then saved into a folder named by file type to be utilized in the follow on code reconstitution.

Identified in the Rovnix sample were the following persistence mechanisms:

- Construction of malicious VBR in conjunction with compressed original one
- Injection of polymorphic bootstrap code;
- Generation of new malicious DLL, titled BKSetup.dll;
- Multiple registry changes across multiple hives;
- Implanting of unsigned driver at end of file system data;
- Hidden partition with backup copies of malware code at end of file system data;

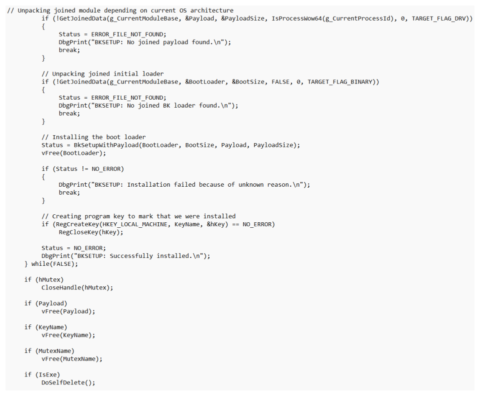
Presented below in Figure 1 is the generation of the boot loader and registry modifications for persistence. In this sample, several persistence creations spawned from a singular source function with the individual modifications completed in their unique functions.

A. Code Reconstitution

Code reconstitution has two phases: (1) Code identification and (2) Code matching and merging.

1) Code Identification and Conversion: Code identification and conversion involves turning this persistence mechanism from the assembly code found in the analysis into source code. The first means to resolve this is by deep searching for the sample’s source code. Approximately twenty percent of the identified persistence mechanisms had source code available. These entries had their code segments that performed the system change for persistence removed, generally consisting of only one or two functions. The code was parred down, removing any repetitive PV value. These code segments were also used as base forms for samples lacking publicly available source code.

A complete comparison was performed against those extracted from the available source code samples, identifying code segments with the same structure. For example, samples within the same family often triggered fifty percent of searches. This allowed the values to be appended together instead of multiple individual entries. Those that do not have a matching structure are marked as new. New entries then have sections of code generated to house the persistence mechanism starting from one of three default templates. One specifically designed for registry changes, with an option presented for the main areas targeted, the second for changes to DLL ordering, and the last for the remainder of system changes.



```
// Unpacking joined module depending on current OS architecture
        if (!GetJoinedData(g_CurrentModuleBase, &Payload, &PayloadSize, IsProcessWow64(g_CurrentProcessId), 0, TARGET_FLAG_DRV))
        {
            Status = ERROR_FILE_NOT_FOUND;
            DbgPrint("BKSETUP: No joined payload found.\n");
            break;
        }

        // Unpacking joined initial loader
        if (!GetJoinedData(g_CurrentModuleBase, &BootLoader, &BootSize, FALSE, 0, TARGET_FLAG_BINARY))
        {
            Status = ERROR_FILE_NOT_FOUND;
            DbgPrint("BKSETUP: No joined BK loader found.\n");
            break;
        }

        // Installing the boot loader
        Status = BkSetupWithPayload(BootLoader, BootSize, Payload, PayloadSize);
        vFree(BootLoader);

        if (Status != NO_ERROR)
        {
            DbgPrint("BKSETUP: Installation failed because of unknown reason.\n");
            break;
        }

        // Creating program key to mark that we were installed
        if (RegCreateKey(HKEY_LOCAL_MACHINE, KeyName, &hKey) == NO_ERROR)
            RegCloseKey(hKey);

        Status = NO_ERROR;
        DbgPrint("BKSETUP: Successfully installed.\n");
    } while(FALSE);

    if (hMutex)
        CloseHandle(hMutex);

    if (Payload)
        vFree(Payload);

    if (KeyName)
        vFree(KeyName);

    if (MutexName)
        vFree(MutexName);

    if (IsExe)
        DoSelfDelete();
```

Fig. 1. Rovnix Registry Bootloader

2) Code Matching and Merging: Once each PV is generated into a code snippet, the elements were pushed into an element of code standardization. Each snippet was labeled via code comments on the type of sample it was extracted from, specifically labeled with sample name and hash. Samples with similar areas of persistence were grouped and sorted to ensure that each value was unique. Duplicates were removed. The process generated a series of white listings containing 800 unique persistence vectors.

VI. Evaluation

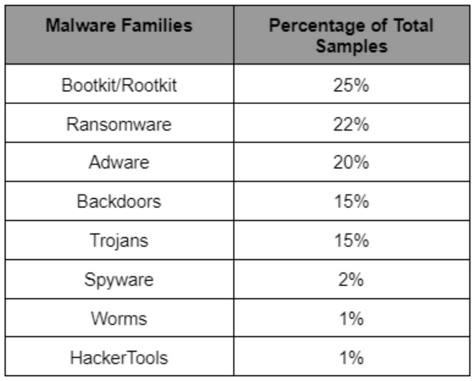
We analyzed the collected and reconstituted persistence vectors above and examine their evolution, complexity, and stealthiness. For evolution, we examined the vectors based on their familial characterization, (e.g., Rootkit, Trojan, Adware, etc). For complexity, we evaluated each sample’s type and the number of persistence mechanisms. Finally, for stealthiness, we assessed their use of obfuscation, such as ease of detection, junk code insertion, and/or the use of encryption.

A. PV Familial Characterization

The largest among the families of the samples was bootkits/rootkits, with just under twenty-five percent of the total samples. The second largest typing was ransomware, with twenty-one percent. Adware was the next largest typing with twenty percent of the total samples. This is due to the transition to tele-networking in recent years, bringing Adware back from among the smallest types to most consistent from 2020. Backdoors and Trojans tied for the third largest typing amongst the samples, each having around fifteen percent of the samples. Worms, hackertool, and spyware had the lowest percentages with around one percent each, with more of the samples coming from farther back in history, late 1990 to early 2000. Figure 2 shows this breakdown.

B. PV Type and Complexity

From all the samples, the PVs utilized followed a two fold progression. As the samples grew more modern both the number of persistence and the type changes, thus increasing their complexity. Older samples, up to 2010, generally worked



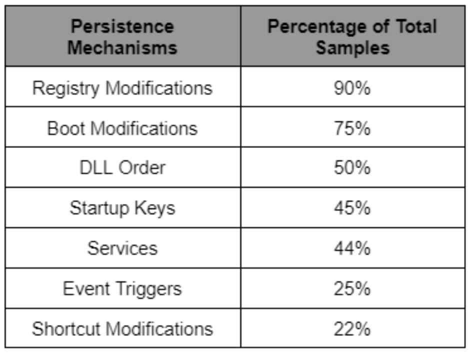
Malware Families	Percentage of Total Samples
Bootkit/Rootkit	25%
Ransomware	22%
Adware	20%
Backdoors	15%
Trojans	15%
Spyware	2%
Worms	1%
HackerTools	1%

Fig. 2. PV Family Characterization

with one established PV per sample. This is most likely due to the limitations of security tools to properly identify the infections on systems. From this they did not require the newer means to ensure their persistence on the system. Modern samples and those dating back to as early as 2017, have started to utilized multiple contingent persistence vectors, allowing for protection of the persistence on the system even if one or two of these is identified. An example of this is the Haxdor-Gen rootkit. As a modular based malware sample, the author is able to tailor the deployment, were included in, as of writing, twelve different persistence vectors. These include generated registry keys, root services, and start up scripts, to name the most common. Included in the source code are commands to reinstall any deleted or removed persistence from the surviving, such as the start up service with code to reinstall an additional copy of the scheduled tasks and to redeploy the virus code into a secure region of the system.

The most common persistence methods utilized by malware are: Registry modification, DLL Replacement/Reorder, startup Keys, Services, and boot modifications [9]. Of the 1000 malware samples studied, all utilized one or multiple of these to establish persistence. We found Registry modifications in most of samples, which was to be expected due to the straightforward ability to generate change. Boot modifications were the next largest of the PVs found amongst the sample base. Consistently, these were found in the more modern malware, as this placed the persistence in areas that are not checked by security tools or are able to start prior and bypass. A small percentage of the older samples did contain this PV, but this was very selective, and found exclusively within the bootkit and ransomware families.

DLL order modifications and startup key were in approximately half of the sample base. However, we noticed none of the hackertools and the spyware families included this PV. Broken down even further, the DLL modifications were a two-to-one in regards to order modifications versus DLL replacement. The more modern malware leaned more on the replacement of the DLL, placing instead its code wrapped in a legitimate version of the DLL. Services were the next largest percentage of the PVs from the sample base. Of the samples there was the common theme that majority were generated



Persistence Mechanisms	Percentage of Total Samples
Registry Modifications	90%
Boot Modifications	75%
DLL Order	50%
Startup Keys	45%
Services	44%
Event Triggers	25%
Shortcut Modifications	22%

Fig. 3. Trend Pattern of PV Type and Complexity

at the Windows system level. Only five percent generated services at the user level, paired with other persistence methods, to launch higher privilege execution. Services PVs were found across majority of malware families, however the largest concentration came from the ransomware, adware, spyware, and rootkit families.

Event triggers were the most diverse PVs, fitting only together based upon the requirement of an action to cause the triggering. Triggers involved various programs being executed, certain accounts being logged in, a specific interrupt, user key inputs, and even screen saver launching. The largest family utilizing event triggers were Trojans with roughly sixty percent implementing at least one event trigger. Least amongst the identified PVs in the sample base was the shortcut modifications. These modifications were generally found in only twenty-two percent of the samples, specifically more in the Trojans, the hacker tool, and portions of the adware. Figure 3 shows a breakdown of all the samples based upon their persistence vectors.

As the age of the samples evolved, the complexity of the malware persistence methodologies improved. Early pieces were only able to manage and maintain one persistence method within their code base. More modern samples, such as our example of Rovnix, can support multiple persistence vectors. These allow for the piece to regain persistence even if one of its vectors is identified and removed.

C. *PV Stealth Factor Categorization*

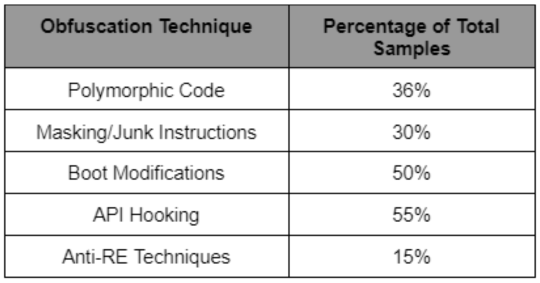
In this analysis, we examined the security elements utilized by our samples' persistence mechanisms. Inverse to the commonality, registry key modification is proven easiest to detect. Multiple tools, such as Regshot which was partly used to identify persistence vectors, could isolate these changes. However, there is the caveat to this detection in the commonality of false positives and negatives. Limited listings show these modifications made from the malware and those made by more legitimate programs. The most challenging persistence modification to identify was the boot modifications, generally the changes created by Rootkit/Bootkits and certain types of Ransomware. These are difficult for both the user and analyst due to their execution prior to most of the OS functionality. One example is Nemsis bootkit, which contained multiple changes to the core operating system elements. One of the key persistence mechanisms is the rewriting of the VBR, which allows it to start before loading the basic operating system elements. The changes reach the point where the bootkit can reapply itself once the hard drive is changed. Finding and cataloging all these changes proved a substantial challenge. Due to their loading prior to the operating system, several took extended static analysis to identify as dynamic analysis could not be relied upon.

Anti Reversing is one of the elements under consistent evolution, with the complexity increasing nearly exponentially as time progresses. Samples with the dates of late 1990s and the early 2000s generally are lacking in complexity of defensive measures. These samples generally had their code as is, due to the lack of diverse options with security tools and the limited knowledge of detecting these samples. These covered the majority of the hackertools and worms. Security improved across the samples with the next section involving masking the sample type within the legitimate functionality. The majority of Trojans and roughly a quarter of the adware samples were predominant in this category. These PVs were masked with generally legitimate changes that would be made to the system, such as with one sample that installed a playable game in conjunction with its malicious payload. While this was a drastic move forward regarding security, the PVs were still straightforward. As with the standard code PVs discussed previously, simple analysis can locate and identify these. Continued progression led to the next level of defensive measures deployed via malware to obfuscate their PVs, covering another twenty-five percent of the adware and roughly fifty percent of the spyware. Segmentation was one of the methodologies in many of the newer samples. Through this process, only a portion of the malicious code is involved in the initially executed malware. Additional elements were requested via system resources once the initial infection was complete. Without a malicious payload, scans of the current code would yield a non-malicious identification.

The final category of defensive measure, predominately found in the newer malware samples, minimizes the items generated to the system's hard drive. This malware evolution has the samples run exclusively on system volatile memory, removing itself once the system is restarted and making it much harder to collect a sample for analysis. None of the samples utilized for the evaluation was this type. Presented in Figure 4 is a breakdown of the stealth functionalities that were found in the sample base. Similar to the complexity of the persistence vectors, the stealth factors evolved exponentially. This stealth is reflexive of the enhancements to security tools designed to catch the common malware attempting to compromise the system.

VII. CONCLUSION AND FUTURE WORKS

This study examines how persistence vectors evolve in complexity and stealthiness over time. It explores the differences in the adaptation of PVs by the different classes of malware, thus paving the way for potential new advancements in malware defense. By shifting focus to these targeted areas for defensive measures, scanning can reduce time, and processor utilization [14]. While not infallible, these persistence scanning elements could be added as an additional layer or decoy in a fully deployed defense-in-depth methodology. Scanning through more diverse operating systems, such as the various ones on Linux and mobile platforms, would be helpful to gain more diverse areas of persistence. Based on this study, our evaluation showed a directed trend in the classification/family of malware away from simple samples like common viruses and evolved into complex multi-module bootkits. Also, we found an exponential trend for complexity and stealthiness, with samples only becoming more adapted to overcome the security tools in place to protect systems. In conclusion, malware is already a significant threat, only increased by persistence, allowing it to remain on the system to perform further malicious activities. Further study of the persistence vectors present across other operating systems could yield similar results. As a recommendation and for future work, persistence utilization can serve as another strong layer for malware prevention in a properly deployed defense in depth.



Obfuscation Technique	Percentage of Total Samples
Polymorphic Code	36%
Masking/Junk Instructions	30%
Boot Modifications	50%
API Hooking	55%
Anti-RE Techniques	15%

Fig. 4. Trend Pattern of PV Stealthiness

REFERENCES

[1] M. Abhijit, and S. Anoop. "Persistence Mechanisms." In Malware Analysis and Detection Engineering, pp. 213-236. Apress, Berkeley, CA, 2020.

[2] Z. Gittins, and M. Soltys. "Malware persistence mechanisms." Procedia Computer Science vol. 176, pg 88-97, 2020.

[3] M. U. Rana, M. Ali-Shah, and O. Ellahi. "Malware Persistence and Obfuscation: An Analysis on Concealed Strategies." In 2021 26th International Conference on Automation and Computing (ICAC), pp. 1-6. IEEE, 2021.

[4] R. Brewer. "Ransomware attacks: detection, prevention and cure." Network security 2016, no. 9, pg 5-9, 2016.

[5] M. Abhijit, and S. Anoop. Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware. Apress, 2020.

[6] N. Virvilis, and D. Gritzalis. "The big four-what we did wrong in advanced persistent threat detection?." In 2013 international conference on availability, reliability and security, pp. 248-254. IEEE, 2013.

[7] R. Anusmita , , and N. Asoke. "Introduction to Malware and Malware Analysis: A brief overview." International Journal 4, no. 10, 2016.

[8] M. O'Leary, and McDermott. Cyber Operations. Apress, 2019.

[9] I. Kirillov, D. Beck, P. Chase, and R. Martin. "Malware attribute enumeration and characterization." The MITRE Corporation, 2011.

[10] L. Zeltser. "Reverse engineering malware.", 2010.

[11] W. Yan, Z. Zhang, and A. Nirwan. "Revealing packed malware." ieee seCurity PrivaCy 6, no. 5. Pg 65-69. 2008.

[12] S. Megira, A. R. Pangesti, and F. W. Wibowo. "Malware analysis and detection using reverse engineering technique." In Journal of Physics: Conference Series, vol. 1140, no. 1, p. 012042. IOP Publishing, 2018.

[13] P. Vinod, R. Jaipur, V. Laxmi, and M. Gaur. "Survey on malware detection methods." In Proceedings of the 3rd Hackers' Workshop on computer and internet security (IITKHACK'09), pp. 74-79. 2009.

[14] R. Tian, I. Rafiqul, L. Batten, and S. Versteeg. ”Differentiating malware from cleanware using behavioural analysis.” In 2010 5th international conference on malicious and unwanted software, pp. 23-30. Ieee, 2010.
[15] R. Al-Shaer, J. M. Spring, and E. Christou. ”Learning the associations of mitre att ck adversarial techniques.” In 2020 IEEE Conference on Communications and Network Security (CNS), pp. 1-9. IEEE, 2020.
[16] C. Kallenberg, S. Cornwell, X. Kovah, and J. Butterworth. ”Setup for failure: defeating secure boot.” In The Symposium on Security for Asia Network (SyScan)(April 2014). 2014.
[17] P. Black, and J. Opacki. ”Anti-analysis trends in banking malware.” In 2016 11th International Conference on Malicious and Unwanted Software (MALWARE), pp. 1-7. IEEE, 2016.
[18] B. Min, V. Varadharajan, U. Tupakula, and M. Hitchens. ”Antivirus security: naked during updates.” Software: Practice and Experience 44, no. 10, pg 1201-1222, 2014.
[19] J. Mankin. ”Classification of malware persistence mechanisms using low-artifact disk instrumentation.” PhD diss., Northeastern University, 2013.
[20] M. S. Webb. ”Evaluating tool based automated malware analysis through persistence mechanism detection.” PhD diss., Kansas State University, 2018.
[21] R. Tahir. ”A study on malware and malware detection techniques.” International Journal of Education and Management Engineering 8, no. 2, vol 20, 2018.
[22] N. Galloro, M. Polino, M. Carminati, A. Continella, and S. Zanero. ”A Systematical and longitudinal study of evasive behaviors in windows malware.” Computers Security vol 113, 2022.
[23] A. Mills, and P. Legg. ”Investigating anti-evasion malware triggers using automated sandbox reconfiguration techniques.” Journal of Cybersecurity and Privacy 1, vol. 1, pg 19-39, 2020.
[24] L. Maffia, D. Nisi, P. Kotzias, G. Lagorio, S. Aonzo, and D. Balzarotti. ”Longitudinal Study of the Prevalence of Malware Evasive Techniques.” arXiv preprint arXiv:2112.11289, 2021.
[25] J. Rutkowska. ”System virginity verifier: Defining the roadmap for malware detection on windows systems.” In Hack in the box security conference. 2005.
[26] Total, V. (2012). Virustotal-free online virus, malware and url scanner. Online: https://www. virustotal. com/en, 2.
[27] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. ”Lest we remember: cold-boot attacks on encryption keys.” Communications of the ACM 52, vol 5, pg 91-98, 2009.
[28] G. Cabau, M. Buhu, and C. P. Oprisa. ”Malware classification based on dynamic behavior.” In 2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 315-318. IEEE, 2016.
[29] V. Khushali. ”A Review on Fileless Malware Analysis Techniques.” vol 9, pg. 46-49, 2020.
[30] J. Kohout, and T. Pevný. ”Unsupervised detection of malware in persistent web traffic.” In 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 1757-1761. IEEE, 2015.
[31] M. Guo, G. Wang, H. Hata, and M. A. Babar. ”Revenue maximizing markets for zero-day exploits.” Autonomous Agents and Multi-Agent Systems 35, no.2. pg 1-29. 2021.