Sterilized Persistence Vectors (SPVs): Defense Through Deception on Windows Systems

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

Aisha Ali-Gombe
Division of Computer Science and Engineering
Louisiana State University, Baton Rouge, AK, USA
aaligombe@lsu.edu

***Abstract*—The vicious cycle of malware attacks on infrastructures and systems has continued to escalate despite organizations' tremendous efforts and resources in preventing and detecting known threats. One reason is that standard reactionary practices such as defense-in-depth are not as adaptive as malware development. By utilizing zero-day system vulnerabilities, malware can successfully subvert preventive measures, infect its targets, establish a persistence strategy, and continue to propagate, thus rendering defensive mechanisms ineffective. In this paper, we propose sterilized persistence vectors (SPVs) - a proactive *Defense by Deception* strategy for mitigating malware infections that leverages a benign rootkit to detect changes in persistence areas. Our approach generates SPVs from infection-stripped malware code and utilizes them as persistent channel blockers for new malware infections. We performed an in-depth evaluation of our approach on Windows systems versions 7 and 10 by infecting them with 1000 different malware samples after training the system with 1000 additional samples to fine-tune the learning algorithms. Our results, based on a memory analysis of pre-and post-SPV infections, indicate that the proposed approach can successfully defend systems against new infections by rendering the malicious code ineffective and inactive without persistence.**

Keywords*— *Malware, Computer Security, Reverse Engineering, Persistence, Rootkit.

I. Introduction

Malware is a continued threat against cyber systems. Characterized by stealthiness, persistence, and mutation, new-generation malware often utilizes various system vulnerabilities for infection and then leverages standard system functionality to maintain persistence. With a suitable persistence strategy, malware can remain active and prolong its existence on a host system. One of the strengths of modern malware development is its adaptability: methodologies mutate rapidly, targeting areas where security measures are weaker or nonexistent. In both related literature and practice, many malware defensive techniques have been proposed - (1) anti-virus and host-based intrusion detection [29], (2) integrity checking [27], [38], detection [7], [24], [36], [37] and (3) after-effect or post-mortem analysis [2], [9], [30], [40], [41] of modern malware. However, as evidenced by the continued rise in stealthier attack scenarios, new samples, and variant development [15], these existing defensive approaches fall short in addressing a growing threat. The common theme of these techniques is identifying the problem either before infection through signature or anomaly detection or after infection through system scans. Neither provides a general means to stop malware due to its adaptability. These ideas of a responsive or reactionary approach to detecting and preventing malware infections, in many respects, play to malware's strengths. Because of the limitations mentioned above, we propose SPVs - a Defense by Deception approach. The goal of our methodology is to drastically reduce malware infections by reducing the available areas of persistence for a malicious actor's exploits, including zero-day attacks. Our approach employs the use of malware code segments to defend a target system against future infection, thus serving as a defensive mechanism. This novel technique is a drastic shift from the conventional utilization of malware code for signature detection and fingerprinting. In our proposed approach, we place blockers called SPVs in critical areas of persistence on target systems. These SPVs are persistence and deployment elements stripped from the various malware samples analyzed. Essentially, SPVs prevent a new malware infection by either blocking it from writing its own vector or overwriting the persistence vector associated with already established malware. With this approach, malware loses its ability to persist and is prevented from executing its payloads, and consequently propagating further. We implemented the prototype of our SPV by manually building a library of 75 payload-stripped SPVs into the Defense by Deception code base, which is then compiled into a target system and deployed at system startup. The Defense by Deception code called the *SPVExec* is then administered as a malware defensive apparatus on a need basis automatically at system runtime without user intervention. The empirical results of the evaluation on Windows 7 and 10 for pre- and post SPV deployment infected of 1000 malware samples showed that the use of SPVs is a very effective strategy for malware defense. For 99% of the samples in the data set, the SPV Defense by Deception process rendered them inert - the malware sets were not able to execute their payloads, persist, or propagate.

Contributions - Our proposed novel SPV strategy provides the following salient features:

- **Defense Against Malware**: The development of a practical approach to preventing new malware infections by simulating and inventing the perception that the system is already infected.
- **Fully Automated Deployment Process**: The deployment and rendering of the SPVs at runtime is done without human intervention.
- **Efficiency**: The SPV code incurs very minimal overhead on run-time system resources.
- **Usability**: The generated SPVs are reliable and seldom flagged as malware by system defense and antiviral tools. Furthermore, the proposed system allows for legitimate programs to be installed without hindrance based upon internal whitelisting.

The rest of the paper is organized as follows: Section 2 presents the problem statement and an overview of rootkit infection; Section 3 provides a detailed description of the SPV process; Sections 4 and 5 present the implementation and evaluation of our research, respectively; Section 6 reviews the related literature; and Section 7 concludes the paper.

II. Related Work

Means of malware detection have grown more stagnant in the last ten years. As shown in Tahir, Alsmadi, and El Merabet [42], [43], [44], most of the improvements have been focused on implementing machine learning. This implementation is worked by classifying individual features within malware samples and rejecting non-specific elements found within a large number of malware samples. While this is an improvement upon the standard malware detection means,

 ISBN: 978-1-61208-996-6

there is the limitation that they are process intensive, both in the means of teaching algorithms for detection also in the scanning of the multitude of files that are presented to the system. The remainder of the detection methodologies can be broken down into Host-based detection, Hypervisor-based detection and Post-mortem analysis.

A. Host-based Detection

The more traditional technique for rootkit detection is a host-based intrusion detection system that checks for anomalies or footprints of known malware. For example, the System Virginity Verifier verifies the validity of in-memory code for critical system DLLs and kernel modules; [35] checks the legitimacy of every kernel driver before it is loaded into the operating system; Panorama [36] is designed to perform behavioral runtime tracking; and SBCFI [22] detects threats by examining the control flow integrity of the kernel code. A smaller subset of methods, such as Autovac, utilizes forensics snapshot comparison engines to detect the execution of malware on the system to prevent it [34]. Other host-based rootkit detection systems include HookFinder [36] and HookMap [32]. These techniques use systematic approaches to detect and remove malware hooks in target operating systems. One major drawback of traditional host-based detection methodologies is the ability of the malicious entity to evade detection, since it is running with the same level of privilege as the detection systems. Since most of these tools are designed to probe for the rootkit signature and/or behavior, malware can easily subvert this effort by hiding its footprint. Malicious actors can employ obfuscation techniques, such as altering the checksums, implementing collection encryption, and setting file wiping [7] to thwart analysis. The SPV code does not scan for malware footprint or traits; instead, it takes the more aggressive approach of hijacking the persistence area of a potential rootkit, leaving the malware with no place to hide. Furthermore, the SPV code is built so that the malware cannot eject or terminate its process.

B. Hypervisor-based Detection

Integrity checking is a technique that requires continuous monitoring of the kernel code for changes to signatures, control flow, and kernel data structures. For kernel-level rootkits, the most practical approach for maintaining kernel integrity is hypervisor-based systems that leverage virtual machine introspection (VMI) [1], [13], [14], [24], [26], [27], [38], [39]. VMI systems and tools are built to introspect the virtual environment through the hypervisor. Since the hypervisor runs at a much lower level than the virtual OS, these mechanisms are often seen as an effective means of detecting rootkits and monitoring their behavior. However, their major limitation is the fact that they target only virtualized environments and cloud infrastructures and cannot be applied to introspect real hardware-based systems. Moreover, most kernel integrity-check-based systems are susceptible to return-oriented rootkit attacks [13]. Methods used to detect the integrity of a system have been proven to be limited based upon the existence of UEFI bootkits. These malicious code elements work by making the operating system accept that malicious code pieces are a legitimate portion of the system's code [8], [12], [23], [33]. With our proposed SPV Defense by Deception process, the system is designed to execute on both hardware and virtual systems, thus circumventing this limitation.

C. Post-mortem Analysis

The last category of rootkit detection methods is postmortem analysis systems, designed to analyze the after-effects of rootkit execution. These forms of analysis are often passive and involve examining kernel memory snapshots looking for evidence of rootkit infection, persistence, and stealth. Disk forensics tools, such as [2], [9], [30], [40] are used for general system incident response. These tools can examine a target system for file modifications, running processes, network activities, and more. In much the same way as integrity checkers, disk forensics tools are limited by their coverage. If malicious code hides its elements in specific system files or structures, these will generally be missed by the aftereffect analysis [6]. With memory forensics, aftereffect analysis is carried out on a snapshot of volatile memory. The most widely used memory analysis framework is the volatility framework [41]. This methodology is restricted to current events and processes. Terminated malware behaviors cannot be retrieved. Furthermore, modern rootkits can evade detection from memory forensics tools by performing direct kernel object manipulations that hide their presence from registering in major kernel structures or by altering the memory collection or imaging process as a whole [17]. In comparison to a more passive malware detection approach, our SPV process is an offensive approach that prevents malware infections in real time. The SPVs are designed to block malware from executing, thus forcing the malware to terminate its process.

D. Problem Statement

Malware has always had the strength of its adaptability, which enables it to use multiple mechanisms to infect and evade detection or bypass many of the elements of system defense [11]. Either through using out-of-date signatures, exploiting unknown vulnerabilities, or targeting the weakest link - the human - malware will cause the defense to fail, even if only one of these falls short. Current detection and prevention tools are at a significant disadvantage in that malware is evolving at a much faster rate than defense tools. Stealthy zero-day attacks are becoming increasingly common, and it takes only a single unknown offense or human error to bring down the whole gauntlet of defenses [16].

Thus, we present the SPV Defense by Deception process - a novel technique that attempts to hijack the areas in which malware in general and rootkits in particular can land their persistence vectors. Rootkit persistence vectors are specifically selected in this research because they are the most common persistence mechanism used by malware of all families [11].

The motivation to use persistence vectors stems from the fact that, in practice, infection vectors are unpredictable, meaning that exploits, especially zero-day exploits used to launch malware attacks, evolve with newly found vulnerabilities. However, the persistence vectors with which the malware maintains a presence on a victim's machine are often deterministic. As such, the most effective way to curtail rootkit infections and ultimately render them ineffective is to place blockers in the potentially persistent channels in the system. Long-term malware campaigns, specifically those utilized by Advanced Persistent Threats (APTs), do not wish to bring a targeted system down immediately. Instead, they wish to complete target profiling against the network, exfiltrate sensitive data, and work further into the system. It can sometimes be months before the threat actors launch their final attack target. For this, they require a means to remain in the system. They require persistence. One of the longest of these types of campaigns was the Harkonnen Operation. Malicious actors could utilize their malware persistence and operate on a network for twelve years before they were finally detected. During this time, the malware implanted could assist with further target development, stealing essential data, such as corporate financial documentation, and pilfering money for the attackers [19]. Our approach injects the SPV code into the system startup process and can be rendered on both bare hardware and virtualized environments. The SPV process blocks all malware by first detecting in real-time when the malware deploys its persistence vector. It then hijacks the malware area of persistence by automatically selecting and overwriting the malware code with certain SPVs. This process consistently blocks target malware from maintaining a presence on a defended system. Although our approach is currently limited to the categories of malware containing persistence vectors, "fileless" malware has only existed substantially since 2002. It is still not utilized as substantially as persistent malware [20], thus making this limitation minimal.

 ISBN: 978-1-61208-996-6

III. The SPV - Defense by Deception Process

The SPV process is a code implementation of "sterilized" malware, or malware that has had its malicious content removed, injected via a common infection mechanism. It is a technique designed to prevent malware persistence on a system. SPV process involves injecting a malware persistence vector into a clean system to block potential malware from maintaining access. This process requires combining standing entries consisting of stripped malware persistence vectors and infection code fragments with filler code. With SPVs, the malicious payload code fragments are entirely stripped off while retaining the core elements of malware, such as API hooking, process manipulation, and service control in the SPV. The workflow for our proposed approach is made up of the SPV development phase and SPVExec code deployment and integration.

A. Development Phase

This phase begins with the identification and extraction of malware persistence vectors, followed by the reprogramming of the extracted persistence code fragments into one executable module.

1) Persistence Extraction: The mechanism in this stage requires manual extraction through detailed reverse engineering. We completed our reverse engineering via both static and dynamic malware analysis techniques. Malicious samples were collected from virus repositories: VirusShare [1] and Malshare [21]. One thousand samples were run through the two phases of reverse engineering. This was completed in a series of virtualized environments of the Windows operating system: Windows 7 and 10, with two copies of each, one for dynamic analysis and one for static analysis. Each machine had two 2.4 GHz cores and 4 GB RAM. For each target malware, we ran the sample against an unpacker to remove any possible common packers and cryptors, leaving behind the bare-bones malware code that would be evaluated by the analysis tools. In this initial phase, the stripped malware code was executed in a custom-built dynamic analysis sandbox running ProcMon, CaptureBat, CFF Explorer, API Monitor, and RegShot.

This static analysis identifies specific part of the executable targeted during the dynamic analysis phase. Such code construct include specific API invocation, non-normal network traffic, registry modification, and file creation. We executed the samples through a debugger and disassembler for the dynamic analysis, specifically IDAPro and OllyDbg, targeting the identified elements in static analysis. Then through the utilization of the HexRay program within IDAPro, the code section was removed and converted to a C program snippet.

2) SPV Generation: With the elements of persistence and infection identified and removed from the base malware code, we developed the SPVs. Since the identified persistence code was disassembled, we began this stage by converting the assembly code into C programming language.

PVs upon extraction reflect specifically that individual sample of the malware, but additionally can be utilized against the majority of the samples of that specific malware family of that generation. For example, an extracted persistence vector from Zeus Botnet would identify not only that specific file but also the different samples in that same generation of Zeus. Specific PVs could also be utilized against other families, dependent upon source code sampling utilized by the author upon its creation. Prior or future versions would require additional PV extractions depending on the evolution of the malware sample.

Figure 1 shows the PV extracted from Necurs Rootkit. The Necurs sample persists using multiple techniques but notably the implementation of boot and registry modification. These specific PVs were identified through our two-phased reverse engineering and exported for inclusion in the SPV library.

These 800 individual SPV extracted from the 1000 malware samples are loaded into the SPV Defense, including the deployment

```
    // 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. Extracted PV

code elements. These were selected as they covered the range of persistence vectors and allowed for broad defense of the SPVs when deployed on the system.

To build a stronger SPV defensive process, we developed an SPV library consisting of a combination of multiple SPVs.

B. SPVExec Deployment and Integration

The proposed SPV mechanism uses these extracted PVs to form a benign rootkit of the SPV and implements persistence elements in the areas extracted in the SPV code called the SPVExec. Additional persistence scanning mechanisms, like the Wingbird scanning ability for its infections, were added to the code to overwrite non-whitelisted persistence modifications. Additionally, previously removed malware functionality deployed a FAT32 file system within the bootstrap code section was added to the system. This area was used for SPV library, whitelisting, and the SPV Defense base code. The data remained encrypted, utilizing a 256-bit key to protect against registering on scans. The SPVExec was implemented as a single Windows executable program loaded alongside the essential boot files at system startup. The prototype is approximately 1800 lines of code in the C programming language. The code is a collection of SPVs, filler code consisting of protective measures extracted from malware, dynamic white- and blacklisting, the learning algorithm, and the SPV launcher.

1) Infection Code Scanning and Rewriting: After successful loading of the SPVExec, the persistence vectors employ two scanning techniques to validate and ensure that an intruder has not altered the injected SPVs at runtime. The first check utilizes time-based scans, similar to those employed by current protective tools. In the current implementation, this check runs a scan every second. Our secondary scanning technique leverages API hooking to check for malware intrusion. The SPV instances are injected into kernel-level processes. Any attempts to access the protected area of persistence are redirected to one of the SPV Defended DLLs. Both scanning techniques utilize hash lookups. During SPV code deployment, a hashmap of the injected SPVs and the region of persistence are stored. The rewriters dynamically replace code elements within the SPVExec

 ISBN: 978-1-61208-996-6

codebase and are designed to look up any changes to the injected SPVs. The dynamically computed hashes of the injected vectors are then compared against the hashes of the SPVs that are expected to be at those regions. If any of the values return no match, then the code rewrites those SPVs as expected.

IV. Evaluation of the SPV Defense by Deception Process

We evaluate the effectiveness of our proposed SPV defense mechanism by performing four major experiments that answered the following questions:

- **Persistence of the SPV Defense process -** Can the SPV Defense survive and persist through system restarts and power removal?
- **Defense against malware -** Can the SPVs be used as an effective strategy to block potential malware from writing to protected areas of persistence?
- **Defense Through Deception -** Does the SPV Defense identify as malware to other malware and legitimate to legitimate programs?
- **System Performance -** Can the SPV Defense process be used as an efficient apparatus for system defense without depleting system resources?
- **White Listing Capability -** Does the SPV Defense allow legitimate programs to install without being replaced with SPV code?

A. Test Environment

To test SPVs across operating systems, we generated Testbed-3 and Testbed-4, utilized Windows testbeds using the same baseline operating systems as in the persistence extraction phase, i.e., Windows 7 and Windows 10. They both contain sets of virtual machines and bare metal with two 2.4 GHz cores and 4 GB RAM. Testbed-1 remained at the same level of security as that of the persistence extraction environment; this removes the chance of malware failing to infect because of patching or security tools. Unlike in persistence extraction, however, this testbed has most of its nonsecurity functionality restored. This allows the system to act similarly to a standard user system that would be part of a normal network. Testbed-2, Testbed-3 and Testbed-4 are equipped with system security monitoring tools, such as operating system inbuilt defense, i.e., Windows Defender, Host-based Security System, and other commercial off-the-shelf antivirus products. For all the testbeds, user programs were installed to simulate a working system that would be on a network and typical applications that are often targeted for compromise. To provide better containment during our analysis and testing, we implemented FakeDNS to resolve any network traffic.

B. Post-Mortem Analysis Environment

We leverage an in-depth analysis of the extracted memory snapshots of the target systems to evaluate the accuracy, resilience, and performance of the overall SPV Defense process. To perform forensic examinations of the memory dumps, we created a separate system equipped with FTK and Volatility. Additionally, to protect the data from being compromised on the system after malware infection, the collection tools were loaded on a USB. This allowed the acquisition to have a limited impact on the system while also keeping the tools from being impacted by any potential built-in anti-analysis approach.

C. Experiments

1) Experiment I: Persistence: Vital to the functionality of the SPVExec benign rootkit is its ability to maintain persistence. To test this functionality, we took the Testbed-2 system post SPV deployment and saved it as "X-Security-TestingPost." We then performed a power cycle. An start up alert was entered into the code to present a popup if the SPV remained in tact. This alert displays the first SPV value and a "Hello World" message. Upon powering the system on, a memory collection was completed utilizing FTK Imager. The memory image was processed by Volatility Memory Framework with the following plugins: psxview, malfind, ldrmodules, apihooks, dlldump, procdump, and threads. Processes and Dynamic Link Libraries (DLLs) of the SPVExec were found that proved that it could maintain its persistence, and a popup was displayed.

Windows 7	True Positive	True Negative	False Positive	False Negative	Overall Accuracy
Symantec	999	0	0	1	99.9%
McAfee	981	0	0	11	98.1%
Kaspersky	987	0	1	12	98.7%
SPV	999	0	1	0	99.9%
Windows 10	**True Positive**	**True Negative**	**False Positive**	**False Negative**	**Overall Accuracy**
Symantec	999	0	0	1	99.9%
McAfee	981	0	0	11	98.1%
Kaspersky	987	0	1	12	98.7%
SPV	999	0	1	0	99.9%

Fig. 2. SPV Evaluation: Regular Testing

2) Experiment II-A: Defense Against Malware: The primary functionality of the SPVExec is its ability to stop malware attacks against the system. To provide a sufficient test of the defensive capabilities of our approach, we conducted this experiment with 1000 malware samples with diverse infection and persistence vectors and varying degrees of stealthiness. We utilized Testbed-2, Testbed-3, and Testbed-4 and executed the SPVExec; the image was saved as "X-Post-SPV," with X representing the OS. Each malware sample was then executed, and a snapshot and memory collection were taken. The system was then reset with the "Post-SPV" images and infected with the next malware sample. As each memory dump was analyzed with Volatility with the above mentioned plugins, the persistence elements of the SPV were found without the markers of the malware surviving. This proves that the SPV Defense was able to prevent the malware from taking effect and rendered it inert, on the same level as other security tools. Comparisons of our process to standard antivirus software indicated that our proposed approach achieves the same level of accuracy as other COTs anti-viruses as shown in Figure 2.

D. Experiment II-B: Reversion Testing

For this experiment, an additional image was generated of Testbed-2, Testbed-3, and Testbed-4, titled "X-SecurityReversion-TestingPost." The commercial antivirus software had the signature libraries reverted back three iterations, allowing for newer malware to be tested as though it were a zero-day exploit. The sample repository listed above was run on both virtual machines. Compared to standard antivirus detection rates, SPV Defense was able to maintain consistent rates. However, during the zero-day detection experiment, it was able to double the detection rates of standard antivirus software, as shown

 ISBN: 978-1-61208-996-6

Windows 7	True Positive	True Negative	False Positive	False Negative	Overall Accuracy
Symantec	525	0	100	375	52.5%
McAfee	495	0	105	400	49.5%
Kaspersky	500	0	25	475	50.0%
SPV	999	0	1	0	99.9%
Windows 10	True Positive	True Negative	False Positive	False Negative	Overall Accuracy
Symantec	525	0	100	375	52.5%
McAfee	495	0	105	400	49.5%
Kaspersky	500	0	25	475	50.0%
SPV	999	0	1	0	99.9%

Fig. 3. SPV Evaluation: Regression Testing

in Figure 3. This proves that SPV Defense is capable of performing far better than commercial malware detection tools against unknown threats due to its targeting only the persistence vectors.

1) Experiment III: Deceptive Capability: For this experiment, the SPVExec was run against two unique phases. One phase determined if malware identified SPVs as similar malware, avoiding infections. The second is if legitimate programs, such as Antivirus, saw the SPVs as a benign code structure. For defense through deception testing, the system was reverted to a save of the SPV defended state presented in Testbed-1. The Necurs malware sample was run against the system. This particular sample was chosen because it has a built-in function searching for already modified keys signaling an infected system. A total of ten instances of the malware were executed in attempts to infect the system; each time, memory collections were completed. Upon analysis of the memory samples via the Volatility analysis, no signs of the Necurs malware were present. Benign testing was conducted against a pool of fifteen antiviruses that ran against the SPV code base. All tests returned negative, indicating that none of the antiviruses flagged the SPVs as malicious.

2) Experiment IV: System Performance: In this experiment, we evaluate the effectiveness of our approach on system resources, particularly the impact of the SPV Defense process on memory and CPU utilization.

(i) CPU Utilization: Utilization was recorded in two separate instances to obtain a baseline for the pre- and postdeployment system. Baseline scores for each of these system performances were recorded. Next, multiple applications were opened to simulate a typical user's desktop, including two Microsoft Word documents, a single instance of Google Chrome, and one instance of the Windows file structure. The system was then left under these conditions for a period of 10 minutes. In the same way as most effective rootkits perform malicious activities without overloading the system, SPVs run in the background without exhausting CPU resources. The CPU usage overhead is on par with that of average antivirus software or an IDS/IPS, which is approximately 2 percent on average [29].

(ii) Memory Utilization The amount of memory utilized by the SPVs, specifically as they spawn processes, is also crucial. Too much memory utilization can cause an internal denial of service, making the method unusable. Utilizing the same parameters as in the CPU overhead test, the system was run with the same software instances for the 10-minute implementation. The baselines were again compared. This result also showed minimal impact on the system resources.

3) Experiment V: White Listing Capability: All the experiments conducted above were able to prove the ability of the proposed method to block future malware infections. However, this would be moot if normal programs were unable to make low-level system modifications and maintain their persistence. For this experiment, we attempted to install 10 "legitimate" programs on an SPV Defended system and determined that all were still installed after system restart. These programs were PyCharm, Visual Studio, BitRise, Atom, BlueFish, CodePen, Crimson Editor, Eclipse, Komodo Edit, and NetBeans. Each of these software programs was examined by the same methodology as malware to determine the major system changes made to ensure their own persistence. Individual snapshots from the "X-Post-SPV" series had one of the above ten programs installed. Memory collection was completed, and a snapshot was taken, titled "XPost-SPVTool", with X being the software installed. Upon powering on, a second memory collection was completed. Finally, the application was tested for functionality by launching the program. In all instances, both the SPV Defense and the program were operational and maintained persistence.

V. Conclusion and Future Works

In this paper, we present a new SPV Defense by Deception strategy that leverages sterilized persistence vectors extracted from a real malware corpus to block potential malware infections. Our system utilizes code from malware samples, not as signatures but as defensive strategies that stop new infections from attempting to write into persistence regions. Compared to existing COTs and techniques described in the literature for malware detection and prevention, our approach is designed to be more robust and versatile, with the ability to block malware both on bare hardware and in virtualized environments. Additionally, our methodology does not require a signature or agnostic of the target malware behavior. Through an in-depth evaluation of 1000 malware samples with pre- and post SPV infection, we demonstrate that our proposed SPV Defense by Deception mechanism can be used to effectively defend systems against malware infections with 1-3 percent CPU and memory overhead while not limiting the ability to install legitimate programs properly.

While this is a strong defense against malware implementation, it is currently limited to Windows OS. Additional work can be conducted into the persistence vectors that are different and unique to other OSes, which could prove beneficial, especially in the Unix-based system, as this portion of the computing world is expanding greatly due to the Internet of things, which bulk have some flavor of Unix driving them.

References

[1] I. Ahmed, A. Zoranic, S. Javaid, and G.G. Richard III. "Modchecker: Kernel module integrity checking in the cloud environment". In2012 41st International Conference on Parallel Processing Workshops 2012 Sep 10 pp. 306-313. IEEE.

[2] D. Byers and N. Shahmehri. "A systematic evaluation of disk imaging in EnCase® 6.8 and LinEn 6.1". Digital Investigation. 2009 Sep 1;6(1-2):61-70.

[3] Z. Gittins and M. Soltys. "Malware persistence mechanisms". Procedia Computer Science. 2020 Jan 1;Vol.176. pp. 88-97.

[4] M.U. Rana, M.A. Shaha, and O. Ellahi. "Malware Persistence and Obfuscation: An Analysis on Concealed Strategies". In2021 26th International Conference on Automation and Computing (ICAC) 2021 Sep 2 pp. 1-6. IEEE.

 ISBN: 978-1-61208-996-6

[5] B.V. Prasanthi. “Cyber forensic tools: a review”. International Journal of Engineering Trends and Technology (IJETT). 2016;Vol.41(5). pp.266-271.

[6] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. “Mapping kernel objects to enable systematic integrity checking”. InProceedings of the 16th ACM conference on Computer and communications security 2009 Nov 9 pp. 555-565.

[7] E. Chan, S. Venkataraman, F. David, A Chaugule, and R. Campbell. “Forenscope: A framework for live forensics”. InProceedings of the 26th Annual Computer Security Applications Conference 2010 Dec 6 pp. 307-316.

[8] B.N. Flatley. ”Rootkit Detection Using a Cross-View Clean Boot Method”. AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH GRADUATE SCHOOL OF ENGINEERING AND MANAGEMENT; 2013 Mar 1.

[9] S.L. Garfinkel. “Automating disk forensic processing with SleuthKit, XML and Python”. In2009 Fourth International IEEE Workshop on Systematic Approaches to Digital Forensic Engineering 2009 May 21 pp. 73-84. IEEE.

[10] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu. “Face-change: Application-driven dynamic kernel view switching in a virtual machine”. In2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks 2014 Jun 23 pp. 491-502. IEEE.

[11] I.U. Haq, S. Chica, J. Caballero, and S. Jha. “Malware lineage in the wild”. Computers Security. 2018 Sep 1; Vol.78. pp.347-63.

[12] O.S. Hofmann, A.M. Dunn, S. Kim, I. Roy, and E. Witchel. “Ensuring operating system kernel integrity with OSck”. ACM SIGARCH Computer Architecture News. 2011 Mar 5; Vol.39(1). pp. 279-290.

[13] R. Hund, T. Holz, and F.C. Freiling. “Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms”. InUSENIX security symposium 2009 Aug 10 pp. 383-398.

[14] X. Jiang, X. Wang, and D. Xu. “Stealthy malware detection through VMM-based ‘out-of-the-box’semantic view”. In14th ACM Conference on Computer and Communications Security (CCS), Alexandria, VA (November 2007) (Vol. 10, No. 1315245.1315262).

[15] A. Kapoor and R. Mathur. “Predicting the future of stealth attacks”. InVirus Bulletin Conference 2011 Oct pp. 1-9.

[16] J.D. Kornblum and ManTech CF. “Exploiting the rootkit paradox with windows memory analysis”. International Journal of Digital Evidence. 2006;Vol. 5(1). pp. 1-5.

[17] T.K. Lengyel, S. Maresca, B.D. Payne, G.D. Webster, S. Vogl, and A. Kiayias. “Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system”. InProceedings of the 30th annual computer security applications conference 2014 Dec 8 pp. 386-395.

[18] L. Litty, H.A. Lagar-Cavilla, and D. Lie. “Hypervisor Support for Identifying Covertly Executing Binaries”. InUSENIX Security Symposium 2008 Jul 28. Vol. 22, p. 70.

[19] R. Luh, S. Schrittwieser, and S. Marschalek. “TAON: An ontology-based approach to mitigating targeted attacks”. InProceedings of the 18th International Conference on Information Integration and Web-based Applications and Services 2016 Nov 28 pp. 303-312.

[20] D. Patten. The evolution to fileless malware. Retrieved from. 2017.

[21] Malshare. www.malshare.com. (2019, October).

[22] N.L. Petroni Jr and M. Hicks. “Automated detection of persistent kernel control-flow attacks”. InProceedings of the 14th ACM conference on Computer and communications security 2007 Oct 28 pp. 103-115.

[23] F. Raynal, Y. Berthier, P. Biondi, and D. Kaminsky. “Honeypot forensics”. InProceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004. 2004 Jun 10 pp. 22-29. IEEE.

[24] R. Riley, X. Jiang, and D. Xu. “Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing”. InInternational Workshop on Recent Advances in Intrusion Detection 2008 Sep 15 pp. 1-20. Springer, Berlin, Heidelberg.

[25] J. Rutkowska. ”System virginity verifier: Defining the roadmap for malware detection on windows systems”. InHack in the box security conference 2005 Sep 28.

[26] M. Schmidt, L. Baumgartner, P. Graubner, D. Bock, and B. Freisleben. “Malware detection and kernel rootkit prevention in cloud computing environments”. In2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing 2011 Feb 9 pp. 603-610. IEEE.

[27] A. Seshadri, M. Luk, N. Qu, and A. Perrig. “SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes”. InProceedings of twenty-first ACM SIGOPS symposium on Operating systems principles 2007 Oct 14 pp. 335-350.

[28] M.I. Sharif, W. Lee, W. Cui, and A. Lanzi. “Secure in-vm monitoring using hardware virtualization”. InProceedings of the 16th ACM conference on Computer and communications security 2009 Nov 9 pp. 477-487.

[29] O. Sukwong, H. Kim, and J. Hoe. “Commercial antivirus software effectiveness: an empirical study”. Computer. 2011 Mar 1; Vol. 44(03). pp. 63-70.

[30] S. Vömel and H. Lenz. “Visualizing indicators of Rootkit infections in memory forensics”. In2013 Seventh International Conference on IT Security Incident Management and IT Forensics 2013 Mar 12 pp. 122-139. IEEE.

[31] J. Wang, A. Stavrou, and A. Ghosh. “Hypercheck: A hardware-assisted integrity monitor”. InInternational Workshop on Recent Advances in Intrusion Detection 2010 Sep 15 pp. 158-177. Springer, Berlin, Heidelberg.

[32] Z. Wang, X. Jiang, W Cui, and X. Wang. “Countering persistent kernel rootkits through systematic hook discovery”. InInternational Workshop on Recent Advances in Intrusion Detection 2008 Sep 15 pp. 21-38. Springer, Berlin, Heidelberg.

[33] M. Xu, X. Jiang, R. Sandhu, and X. Zhang. “Towards a VMM-based usage control framework for OS kernel integrity protection”. InProceedings of the 12th ACM symposium on Access control models and technologies 2007 Jun 20 pp. 71-80.

[34] Z. Xu, J. Zhang, G. Gu, and Z. Lin. Autovac: Automatically extracting system resource constraints and generating vaccines for malware immunization. In2013 IEEE 33rd International Conference on Distributed Computing Systems 2013 Jul 8 pp. 112-123. IEEE.

[35] J. Rutkowska. “System virginity verifier: Defining the roadmap for malware detection on windows systems”. InHack in the box security conference 2005 Sep 28.

[36] H. Yin, Z. Liang, and D. Song. “HookFinder: Identifying and understanding malware hooking behaviors”.

[37] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. “Panorama: capturing system-wide information flow for malware detection and analysis”. InProceedings of the 14th ACM conference on Computer and communications security 2007 Oct 28 pp. 116-127.

[38] H.A. Lagar-Cavilla and L. Litty. “Patagonix: Dynamically Neutralizing Malware with a Hypervisor”.

[39] Y. Oyama, T.T. Giang, Y. Chubachi, T. Shinagawa, and K. Kato. “Detecting malware signatures in a thin hypervisor”. InProceedings of the 27th Annual ACM Symposium on Applied Computing 2012 Mar 26 pp. 1807-1814.

[40] O. Vermaas, J. Simons, and R. Meijer. “Open computer forensic architecture a way to process terabytes of forensic disk images”. InOpen Source Software for Digital Forensics 2010 pp. 45-67. Springer, Boston, MA.

[41] A. Mohanta and A. Saldanha. “Memory Forensics with Volatility”. InMalware Analysis and Detection Engineering 2020 pp. 433-476. Apress, Berkeley, CA.

[42] R. Tahir. “A study on malware and malware detection techniques”. International Journal of Education and Management Engineering. 2018 Mar 1; Vol. 8(2). pp. 20.

[43] N. Idika and A.P. Mathur. “A survey of malware detection techniques”. Purdue University. 2007 Feb 2;Vol. 48(2). pp. 32-46.

[44] H. El Merabet and A. Hajraoui. “A survey of malware detection techniques based on machine learning”. International Journal of Advanced Computer Science and Applications. 2019; Vol. 10(1).

 ISBN: 978-1-61208-996-6