An Advanced Memory Introspection Technique to Detect Process Injection and Malwares of Varied Types in a Virtualized Environment

Darshan Tank, Akshai Aggarwal, Life Senior Member, IEEE, and Nirbhay Kumar Chaubey, Senior Member, IEEE

Abstract—Today’s advanced malware can easily avoid detection by adopting several evasion strategies. Process injection is one such strategy to evade detection from security products since the execution is masked under a legitimate process. Malicious activities are often enforced by injecting malicious code into running processes, which is often undetectable by traditional anti-malware techniques. Various process injection techniques are employed by malware to gain more stealth and to bypass security tools/products. Our main focus in this research work is to propose an entirely out-of-VM approach based on advanced memory introspection to detect process injection of varied types in a virtualized environment. We have implemented a plugin using the open-source Volatility tool and successfully tested it on live VMs and malware-infected memory images. Experimental results show that our model classifies injected memory regions with high accuracy and completeness and has more true positives and fewer false positives when compared to other existing systems/solutions. Our proposed detection approach assures precise and reliable results and exactly pinpoint injected memory regions. Our proposed system detects more malware families and dominates the other approaches in all evaluation metrics.

Index Terms—Advanced Memory Introspection, Detection, Malware Analysis, Memory Image, Memory Region, Process Injection, Security, Virtual Machines, Volatility, Windows

I. INTRODUCTION

Cloud computing has become a dominant computing paradigm over the past several years. Virtualization is a key underlying technology to empower cloud computing. Virtualization creates and runs multiple Virtual Machines (VMs) on a single physical machine using Virtual Machine Monitor (VMM). The most alarming threat to the cloud is virtual machine security. Virtual machines are a primary target for an adversary to acquire unethical access to the organization’s virtual infrastructure. As malware becomes increasingly more persistent and sophisticated, traditional security countermeasures and technologies employed for securing virtualized environments are not competent for advanced malware detection. Despite recent advances in malware detection capability, it is still possible for a potential adversary to stay unfound. Process injection is one such strategy to evade detection from security products since the execution is masked under a legitimate process. The motive of this paper is to propose methodologies to detect malware in the form of process injection exploitations in virtual machine's memory. It focuses on eight different process injection techniques: Remote DLL (Dynamic Link Library) Injection Via CreateRemoteThread and LoadLibrary, Remote Thread Injection Using CreateRemoteThread, PE (Portable Executable) Injection, Reflective DLL Injection, Hollow Process Injection, Thread Execution Hijacking, APC (Asynchronous Procedure Call) Injection, and AtomBombing. The malwares are executed on Windows 7 and Windows 10 virtual machines. Dynamic malware analysis is performed using the Volatility framework and LibVMI python bindings.

A. Motivation

Today’s advanced malware utilizes different process injection techniques to either manipulate other running processes or to hide its existence. There exists no single system which has the capability to detect all major types of process injection. Current process injection detection tools are not able to cope with the existing injection techniques and fail to reliably reveal existing malware utilizing certain hiding techniques [1].

This work proposed a novel dynamic malware analysis framework using advanced memory introspection to detect process injection of varied types in a virtualized environment. Although there are numerous process injection techniques, this work focuses on the detection of eight different implementations of process injection: namely, Remote DLL Injection Via CreateRemoteThread and LoadLibrary, Remote Thread Injection Using CreateRemoteThread, PE Injection, Reflective DLL Injection, Hollow Process Injection, Thread Execution Hijacking, APC Injection, and AtomBombing. Instead of detecting infected processes, we aim to detect an actual malicious memory region within the infected process thereby pinpoint the malware exactly.
B. Contributions

The contributions of this paper are as follows.

1) An algorithm which scans memory of VMs to locate malicious/injected memory regions.

2) A Volatility plugin (ProcInjectionsFind) that implements the algorithm and prints information about the injected/victim process’s VADs, hex dump and disassembly information at VADs base address.

3) One can perform live introspection of running VMs for possible indication of process injection.

II. RELATED WORK

Process injection is a versatile technique that facilitates a wide range of actions. Process injection techniques have broad domains and account for many different kinds of behaviors. Attackers and malware frequently employ various process injection techniques. Process injection enables adversaries to get away from defensive controls by executing potentially malicious code in the context of apparently gentle ones. Executing code in the context of another kind of process may also allow access to that process's memory, system or network resources, and likely elevated rights.

There are many different variants of process injection techniques - DLL Injection, Thread Injection, PE Injection, Hollow Process Injection, Reflective DLL Injection, APC Injection, and Thread Execution Hijacking, etc. As there exist no direct access mechanism to the physical memory of VMs in a virtualized environment, detection of process injection is very challenging on a VM as compared to a physical machine.

There exist many different approaches for the detection of process injection, such as monitoring process, Windows API (Application Programming Interface) calls, DLL/PE file events, system calls, and named pipes, etc. There is a case where benign use of Windows API calls may be common and difficult to distinguish from adverse behavior. Most valid processes should not need to use Windows APIs such as CreateRemoteThread, so it is characterized as a very suspicious API and is detected by many security products which also detect the malicious DLL on the disk [2]. The analytics that produced the majority of false positives came from looking for CreateRemoteThread calls from any process. Windows API calls that can be used to alter memory within another process, such as VirtualAllocEx and WriteProcessMemory, may be utilized for the possible indication of process injection [2]. Classic injection techniques can be caught using API call sequence patterns such as – OpenProcess → VirtualAllocEx → WriteProcessMemory → CreateRemoteThread [3].

Various methods exist for discovering possibly injected memory regions inside the virtual address space of the injected process. One such method is the well-known plugin called Malfind for the Volatility memory forensics framework. Volatility’s malfind plugin can detect injected memory sections [4]. Different types of process hollowing techniques get discovered with hollowfind volatility plugin [5]. However, these approaches have significant shortcomings. Malfind be affected by a high false-positive rate, and hollowfind discloses only a subgroup of all admissible types of host-based code injection attacks. Membrane [6] reveals affected processes in place of actually malicious memory regions within a process. Although this decreases the size of the haystack to search in, it does not locate the malware precisely. Another method is the PowerShell Script Get-InjectedThread.ps1 by Jared Atkinson. Get-InjectedThread [7] scans each running thread to conclude if it is the result of memory injection. So, in summary, there exist several systems or solutions to detect process injection attacks in memory dumps, but they all suffer from following major limitations.

- Bypass common detection methods.
- Detects memory regions with execute permission.
- Detects memory regions marked as private.
- Detects only a subgroup of process injections.
- Detects only a single injected code region.
- Unsatisfying detection rates, i.e., high false positive rate.
- Too coarse detection granularity, i.e., detects affected processes in place of actually injected memory regions within a process.
- Some of the existing plugins work correctly on memory images of specific Windows versions and may not work properly on 64-bit Windows guests, i.e., depends on hardware architecture.
- There exists no single system/solution which has the capability to detect all major types of process injection.
- Current process injection detection tools are not able to cope with the different implementations of process injection techniques and fail to accurately uncover existing malware employing certain hiding techniques.

Process injection detection systems could produce a notable amount of data. The produced data might not be directly beneficial for defense unless gathered under particular circumstances [8]. We still need more context to create an operational detection system.

III. THREAT MODEL & ASSUMPTIONS

A. Threat Model

- This work focuses on process injection exploits inside the virtualization environment.
- Our study emphasis on Windows VMs because it is still the ultimate focused operating system by malicious software as reported in [9, 10, 11] and has a larger attack surface compared to Linux systems.
- Our expected attack environments are virtualization environments such as those utilizing the Infrastructure as a Service (IaaS) cloud model, where the host operating system has limited-to-no control over its guest system functioning.
- This work set protective attempt on a detection approach and the detection granularity is on memory region basis.

B. Assumptions

- No inter-VM attacks exists.
- A Hypervisor or Virtual Machine Monitor (VMM) is safe.
- Empty VAD region (entirely consisting of zeros) is considered to be benign. This helps to minimize false positives.
IV. PROPOSED DETECTION MODEL

In this work, the Kernel-based Virtual Machine (KVM) functioning at the Virtual Machine Monitor (VMM) or hypervisor is set up on host machine. Virtual Machines launched by the KVM hypervisor have Windows 7 and Windows 10 guest OS running on it. LibVMI python bindings integrated with the Volatility framework set up on the host OS. Dynamic malware analysis is performed using the Volatility framework and LibVMI python bindings. Figure 1 shows the architecture of our proposed system.

The malwares are executed on Windows-based VMs in the form of process injection exploitations. After the injection is performed, a physical memory image of the virtual machine’s core is retrieved by creating a dump file so that it can be analyzed later or else we can directly access the memory of a running VM using LibVMI python binding and analyze it for possible indication of process injection in real-time. The VMI-based Process Injection Detection (VMIPID) model scans for injected memory regions that are created as a result of process injection.

V. METHODOLOGY OF RESEARCH

There are numerous different techniques to inject code into a running process and execute it. Although each process injection technique varies in execution, all techniques share one common property – allocating memory in the address space of the victim process. Most cases of injected code allocate a new memory region (VAD) inside the virtual address space of the victim process. This work proposes a novel method for detecting potentially injected memory sections inside a process's virtual address space. Our proposed system scans for injected memory regions that are created as a result of process injection.

Following algorithms have been designed and implemented as a volatility plugin (ProcInjectionsFind) to detect process injection of varied types.

- **Determine Suspicious Processes**

  1: Scan all active processes
  2: Enumerate process’ handles in each active process
  3: Filter process’ handles of type THREAD
  4: Examine a thread not handled or created by the process under which it executes (*)
  5: Append thread’ handle PID and TID in suspicious process list

  (*) The following thread’ handles have been excluded.
  - Handles created by csrss.exe
  - Handles created by its parent process

  - csrss.exe is associated with the formation of every process and thread (with the exception of itself and the processes that begin before it) [12].
  - A parent process may create the handle of type THREAD in its child process for an authentic reason.

- **Algorithm 1: To detect “Remote DLL injection Via CreateRemoteThread and LoadLibrary”**

  1: Scan all active processes
  2: Enumerate all threads in each active process
  3: Found thread id in suspicious process list
  4: Check if thread is suspended → Thread’s State is ‘Waiting’ and WaitReason is ‘Suspended’
  5: Traverse process’ VADs and perform the following check on memory regions (VAD) a. Any VAD region marked as private, has a VadS tag, and with execute permission.
  6: Mark the corresponding memory region as suspicious
  7: Dump an entire VAD related to a suspicious memory region
  8: Cross-check with VirusTotal score of the dumped VAD to confirm injection

- **Algorithm 2: To detect “Thread execution hijacking”**

  1: Determine suspicious processes
  2: For each thread in suspicious process list
  3: Relate thread with its corresponding VAD → Check for the file mapped on disk → Thread is mapped to kernel32.dll
  4: Scan thread’s execution for LoadLibrary (or LoadLibraryEx) API function
  5: Correlate DLLs load time with thread create time
  6: Dll’ corresponding VAD node is marked with the value 1 in its NumberOfMappedViews & NumberOfUserReferences fields
  7: Mark the corresponding DLL & memory region as suspicious
  8: Dump an entire VAD related to a suspicious memory region
  9: Cross-check with VirusTotal score of the dumped VAD to confirm injection
- Algorithm 3: To detect following type of injections.
  1) Remote thread injection using CreateRemoteThread
  2) PE injection
  3) Reflective DLL injection
  4) Hollow process injection
  5) APC injection
  6) AtomBombing

The steps of the above algorithms are merged in a volatility plugin known as ProcInjectionsFind, which can be called from Volatility’s prompt. The plugin runs several checks to pinpoint malicious/injected memory regions. A volatility plugin ProcInjectionsFind runs against the Windows memory image or memory of a live VM and examines each memory region to conclude if it is the result of process injection. The described steps are performed on malware-infected memory images and memory of a live VM and the results are confirmed.

Figure 2 shows our proposed system workflow. We carry out experiments on the host machine, which had the configurations shown in Table 1.

### TABLE I

| TESTBED CONFIGURATIONS |
|-------------------------|
| Host OS | Ubuntu 16.04.6 LTS |
| Host OS Type | 64-bit |
| Linux Kernel | Linux 4.15.0-74-generic |
| Architecture | X86_64 |
| Processor | Intel® Core™ i5-8265U CPU @ 1.60GHz x 8 |
| Disk | 1 TB |
| Number of cores & threads | 4 & 8 |
| Physical memory (RAM) | 8 GB |
| Hypervisor (VMM) | KVM |
| Virtual Machine – 1 | OS – Windows 7, vCPU - 1 |
| Memory | Memory – 2 GB, Storage – 40 GB |
| Virtual Machine – 2 | OS – Windows 10, vCPU - 1 |
| Memory | Memory – 2 GB, Storage – 40 GB |
| Tools / Framework used | LibVMI python bindings (version-3.4) & Volatility framework (version-2.6.1) |

### A. Malware Hiding Technique

Malicious executables have been altered to assign the memory region with READ ONLY protection at first and later switch it to EXECUTE_READWRITE. VirtualProtectEx function changes the protection on a memory region in the virtual address space of a stated process [13].

### VI. MODEL EVALUATION AND RESULTS

The proposed VMIPID model scans each process’s memory segments for the presence of injected code and classifies each of these memory regions as being either benign or suspicious. The proposed model runs several checks to spot malicious/injected memory sections. To evaluate the proposed VMIPID model, we used several evaluation metrics. These include precision, recall, false-positive rate, f-measure, and accuracy. The definition and formula of these evaluation metrics can be seen in Table 2.

### TABLE II

| DEFINITION OF EVALUATION METRICS |
|----------------------------------|
| Metrics | Definition |
| True Positive (TP) | Correctly identified injected memory regions. |
| False Positive (FP) | Incorrectly identified injected memory regions. |
| True Negative (TN) | Correctly identified benign memory regions. |
| False Negative (FN) | Incorrectly identified benign memory regions. |
| Metric | Formula |
| Precision (P) | TP / (TP + FP) |
| Recall (R) / Completeness / TPR | TP / (TP + FN) |
| False Positive Rate | FP / (FP + TN) |
| F-measure | 2 PR (P + R) |
| Accuracy | (TP + TN) / (TP + FP + TN + FN) |
 Completeness: The detection system is treated to be complete when all injected memory regions across multiple targeted processes are being discovered.

 Accuracy: Accuracy is defined as correctly identified injected memory regions by a total number of memory regions on the memory image.

VirusTotal offers a free service for scanning suspicious files using several antivirus engines. We have cross-checked our derived results with VirusTotal. Figure 5 shows calculated hash of suspected file. Figure 6 and Figure 7 show the execution of VirusTotal API script. This API script scans a file hash and automatically submits the file hash to VirusTotal and dumps the full VirusTotal report to file. In this way, we don’t require to upload malicious/injected memory regions online for examination.
As depicted in figure 4, procinjectionsfind plugin prints information about the injected/victim process’s VADs, disassembly, and a hex-dump at the VADs base address.

| Sr No | Process Injection Techniques | Compared With |
|-------|------------------------------|---------------|
| 1     | Remote DLL injection         | FindDLLInj [14] |
| 2     | Remote thread injection, PE injection, Reflective DLL injection, Thread execution hijacking, APC injection, and AtomBombing | Malfind [15] |
| 3     | Hollow process injection     | Hollowfind [16], Malfind [15], Threadmap [17], Malfofind [18] |

TABLE III
COMPARISON OF PROCESS INJECTION TECHNIQUES WITH EXISTING APPROACHES

TABLE IV
COMPARISON OF EVALUATION METRICS WITH EXISTING APPROACHES

Fig. 8. Comparison of evaluation metrics with existing detection approaches on win10_VM.

Fig. 9. Comparison of evaluation metrics with existing detection approaches on win7_VM.

Fig. 10. Comparison of evaluation metrics with existing detection approaches on win10_VM.
The experimental results show that the proposed approaches are promising. The derived results reveal several insights. In contrast to Win_7 VM, the proposed VMIPID model reports promising results on Win_10 VM in all evaluation metrics and it also reveals 100% completeness on both Win_7 VM and Win_10 VM. Experimental results show that the proposed model achieves high accuracy and f-measure and has more true positives and fewer false positives when compared to other existing solutions. Overall, the proposed system completely detects more malware families and stands over other systems in all evaluation metrics defined in this work.

VII. CONCLUSION AND FUTURE IMPROVEMENTS

One of the main contributions of this work is to perform live introspection of running VMs for possible indication of process injection. We adopt an Out-of-VM introspection approach thereby monitoring the runtime state of VMs completely outside of VMs, i.e., from the hypervisor. We implemented a prototype called VMIPID as a volatility plugin that can be independently called and runs against the windows memory image or memory of a live VM and examine each memory region to conclude if it is the result of process injection. Our implemented plugin certainly reports any malicious memory region containing injected code, and also evaluates it against...
implementation of hiding technique. Experimental results show that our model classifies injected memory regions with high accuracy and completeness and has more true positives and fewer false positives when compared to other existing solutions.

The proposed framework neither modifies the guest OS at the host level nor influences the performance of the host OS and VMs. The proposed solution detects an actual injected memory region within the injected process and thereby pinpointing the malware exactly. The proposed detection model recognizes injected memory regions into all infected processes. Once the malware is detected within running VMs, security analysts use security measures given in segregation plans.

A. Future Improvements

Our study focuses on eight different implementations of process injection in this work. One could add more implementation and cover other malware hiding techniques as a future work. A developed plugin has been proved against Windows-based VMs, i.e., Windows 7 and Windows 10 VMs respectively. The plugin might be proved against other windows-based operating systems to see its platform compatibility and conformity. Proposed approach could also be ported to Linux-based VMs.

Future work concerns in-depth examination of different operating system’s data structures and new point of view to try different methods to locate injected memory regions that current methods/tools do not reveal. The proposed approach finds memory-only malicious code, i.e., injected code resident in memory. Future work requires deeper analysis to discover injected code stored in the files on disk or scattered all over the paging files.

REFERENCES

[1] Block, F., & Dewald, A. (2019). Windows Memory Forensics: Detecting (Un) Intentionally Hidden Injected Code by Examining Page Table Entries. Digital Investigation, 29, S3-S12.
[2] Hosseini, A. (2017, July 18). Ten process injection techniques: A technical survey of common and trending process injection techniques. Retrieved December 17, 2018.
[3] Y. Qiao, Y. Yang, J. He, C. Tang, and Z. Liu, “CBM: free, automatic malware analysis framework using API call sequences,” in Knowledge Engineering and Management, pp. 225–236, Springer, Berlin, Germany, 2014.
[4] Volatility Foundation: The Volatility Framework (2016). http://www.volatilityfoundation.org. Accessed 24 Apr 2019
[5] Monnappa, K.A.: Detecting deceptive process hollowing techniques using hollowfind volatility plugin (2016). https://cysinfo.com/detecting-deceptivehollowing-techniques/. Accessed 17 Jun 2019
[6] P’ek, G., L’az’ar, Z., V’arnagy, Z., F’elegüh’ azi, M., Butty’an, L.: Membrane: a posteriori detection of malicious code loading by memory paging analysis. In: Askoxyakis, I., Ioamidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9878, pp. 199–216. Springer, Cham (2016). doi:10.1007/978-3-319-45744-4_10
[7] Code from "Taking Hunting to the Next Level: Hunting in Memory" presentation at SANS Threat Hunting Summit 2017 by Jared Atkinson and Joe Desimone - Github. Get-InjectedThread.ps1 https://gist.github.com/jaredatkinson/23905d34537ce4b5b181c3e6405cfd2
[8] Process Injection, Technique T1055 - Enterprise | MITRE ATT&CK ® https://attack.mitre.org/techniques/T1055/, last accessed on 21 October 2019
[9] AVTEST. The AV-TEST Security Report 2016/17. Tech. Rep.; 2017. https://www.av-test.org/fileadmin/pdf/security_report-AVTEST_Security_Report_2015-2016.pdf.
[10] Garnaeva M, Sinitsyn F, Namestnikov Y, Makrushin D, Liskin A. Overall Security_Bulletin_2016_Statistics_ENG.pdf.
[11] Symantec. Internet security threat report 21. https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf.
[12] Michael Hale Ligh, Andrew Case, Jamie Levy, AAron Walters, The Art of Memory Forensics, Wiley, 2014, ISBN: 978-1-118-82499-3
[13] VirtualProtectEx function (memoryapi.h) - Win32 apps. Microsoft Docs (2018, December 5). https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotectex
[14] Balouar S. (November 2018) Master's Thesis on Process Injection Techniques and Detection using the Volatility Framework
[15] The Volatility Foundation, 2017. Volatility's Malfind Plugin. https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/malware/malfind.py.
[16] Monnappa, K.A., 2016. Hollowfind Volatility Plugin. https://github.com/monnappa22/HollowFind.
[17] KSL group, 2017. Threadmap Volatility Plugin. https://github.com/kslgroup/threadmap.
[18] Pshoul, D., 2017. Malfind Volatility Plugin. https://github.com/volatilityfoundation/community/blob/master/DimaPshoul/malfind.py
[19] Injection techniques. Github – theevilibit/injection. https://github.com/theevilibit/injection
[20] Reflective DLL Injection. Github – stephenfewer/ReflectiveDLLInjection. https://github.com/stephenfewer/ReflectiveDLLInjection
[21] Process Hollowing. Github – m0n0ph1/Process-Hollowing.
[22] Seven different DLL injection techniques. Github – fdiskyou/injectAllTheThings. https://github.com/fdiskyou/injectAllTheThings
[23] Process Injection Techniques. Github – scrary/InjectProc https://github.com/scrary/InjectProc
[24] AtomBombing. Github - BreakingMalwareResearch/atom-bombing: Brand New Code Injection for Windows https://github.com/BreakingMalwareResearch/atom-bombing