MIT UNIVERSITY'S - Abhivruddhi Journal
e-ISSN - 2584-1025
VOL.5(01), JUNE, 2025

Ghost in the Endpoint: Techniques for Evading EDR Detection

Anuradha Kadam1, Dr. Ayesha Butalia2, Ashish Jadhav3
1MIT Arts, Design & Technology University, Pune, 2MIT Arts, Design & Technology University, Pune,
1aunuradhask@gmail.com

Abstract

Cyber threats are constantly evolving, and one of the most sophisticated forms of cyberattacks today is fileless malware. Unlike traditional malware that relies on executable files, fileless malware operates directly in a system's memory, leaving little to no trace behind. This makes it extremely difficult to detect and mitigate using conventional security measures. Day by day attackers are inventing new techniques to mute detection technologies. This research investigates the effectiveness of defensive security measures, such as Endpoint Detection and Response and Advanced Antivirus technologies, in identifying and mitigating these advanced threats. Currently Cybercriminals are using fileless scripts due to their ability to bypass defenses. This research reveals potential weaknesses in current security solutions, emphasizing the urgent need for stronger defenses against fileless PowerShell attacks. The insights gained contribute significantly to improving cybersecurity protections against these increasingly complex threats. In this paper, we are discussing techniques used for fileless malware and EDR evasion. Furthermore, we will also discuss ways to detect fileless malware as future scope.

Keywords: MITRE ATT&CK framework, Userland Hooking, EDR, NTDLL, Syscall, kernel, obfuscation, LOLBins, SSN

Introduction

The concept of fileless malware has been around for over a decade, its early forms lacked persistence, disappearing after a system reboot. This changed significantly around 2014 with the emergence of threats like Poweliks, a click-fraud Trojan demonstrating persistent capabilities. Today, fileless techniques are a staple in the arsenals of cybercriminals, posing a significant threat to all organizations due to their ability to evade traditional file-based detection methods. Threat actors are always developing new and more effective approaches to system breaches in the perpetually shifting field of cybersecurity. From basic computer viruses to the sophisticated persistent dangers of today, malware has developed extremely dramatically. According to one report from The Ponemon Institute, fileless malware attacks are roughly ten times more likely to succeed than traditional file-based attacks and Aqua Security's 2023 Cloud Native Threat Report estimates that fileless attacks increased more than 1400% over the previous year. Despite the advancements in security products, threat actors can still leverage some or the other ways to launch successful attacks.

It is confirmed that fileless attacks are growing day by day and its market capture will almost get doubled in next 5 years. To mitigate direct memory manipulations modern EDR comes in picture. It is observed that modern Endpoint Detection and Response (EDR) platforms can identify malicious activity within a process's memory space by injecting user-land hooks into selected Windows API (WinAPI) functions. These hooks are essentially assembly code instructions that redirect the execution of specific threat behaviors to a vendor-owned Dynamic Linked Library (DLL), which is loaded into the user process to determine if the behavior is malicious [11]. This capability is distinct from other EDR functionalities, such as malware detection based on static file signatures, heuristic analysis of files, or other detection methods that do not rely on the WinAPI call chain. This observation gives motivation to study various hook and DLL removal techniques to identify potential detection gaps. This paper focuses on the survey of user-land hooks evasion techniques within processes, and which removal technique is most effective amongst the ones mentioned in this paper. On a system with a modern EDR platform that employs user-land hooks in all user processes, an attacker can execute various MITRE ATT&CK framework exploitation and post-exploitation techniques within processes by leveraging in memory execution where user-land DLLs and hooks have been removed, without detection.

As per common trends among organizations, significant trust is shown in EDR platform vendor efficacy claims and it is believed that absence of EDR alerts is an indication of no malicious activity. This leads to insufficient operational testing at organizational level. These trends create opportunities for attackers to exploit undetected vulnerabilities, particularly when skilled attackers use techniques to remove user-land hooks without detection. This survey highlights common techniques to understand how attackers can evade common defenses, significantly impacting system and network security.

II. Foundations of In Memory Execution And Windows Syscall Orchestration

A. Fileless Attack Flow

Fileless attacks are cyber threats that compromise systems without installing any files. Instead, they use tools and processes already present in the operating system, like PowerShell and Windows Management Instrumentation (WMI). By executing malicious code in-memory and exploiting trusted system components, these attacks avoid detection by traditional antivirus software. This makes it harder to detect and analyze, requiring advanced security measures focused on behavioral analysis and anomaly detection.

Key Characteristics of Fileless Attacks

1. No Malicious Files on Disk: Fileless attacks do not leave behind any malicious files on the victim's system. This makes them invisible to file-based detection mechanisms.
2. Execution via Legitimate Tools: Attackers use trusted system tools like PowerShell, Windows Management Instrumentation (WMI), or Microsoft Office macros to execute malicious commands. Since these tools are part of the operating system, their use does not raise immediate red flags.
4. Resides in RAM: The malicious code runs entirely in the system's memory (RAM). When the system is rebooted, the malware disappears from memory, leaving no trace on the disk.

Bypasses Traditional Defenses: Fileless attacks can bypass firewalls and antivirus software because they do not rely on malicious files or executables that can be scanned or blocked.

Attack Flow of a Fileless Attack

1. Initial Compromise: The attacker sends a phishing email containing a malicious link or attachment. The email is designed to trick the victim into visiting a malicious website or opening an infected document.
2. Exploitation of Vulnerabilities: Once the victim interacts with the phishing email, the attacker exploits vulnerabilities in the victim's web browser or browser plugins (e.g., Flash, Java) to gain initial access to the system.
3. Execution via PowerShell: The attacker uses PowerShell, a powerful scripting tool built into Windows, to execute malicious commands directly in memory. PowerShell scripts can download and run additional payloads, manipulate system settings, or exfiltrate data without writing to disk.
4. Data Collection and Exfiltration: The attacker collects sensitive data from the victim's system, such as credentials, financial information, or intellectual property. This data is then exfiltrated to a remote server controlled by the attacker.

Advantages of Fileless Attacks for Attackers

- Stealth: Fileless attacks are difficult to detect because they do not leave traces on the disk.
- Persistence: Attackers can maintain persistence by leveraging legitimate system tools and processes.
- Evasion: Fileless attacks bypass traditional security solutions like antivirus software and firewalls.

Challenges for Defenders

- 1. Detection: Since fileless attacks do not rely on malicious files, traditional signature-based detection methods are ineffective.
- 2. Forensics: Fileless attacks leave minimal forensic evidence, making it difficult to investigate and attribute the attack.
- 3. Persistence: Attackers can use legitimate tools like scheduled tasks or WMI to maintain persistence, making it hard to completely remove the threat.

One of the mitigation Strategy against in memory execution is using EDR Userland hooking technique

B. Windows Syscall Orchestration

Fig 2 illustrates the flow of a system call from an application through various layers of the Windows operating system, ultimately reaching the hardware. Each layer plays a crucial role in ensuring that the system call is executed securely and efficiently.

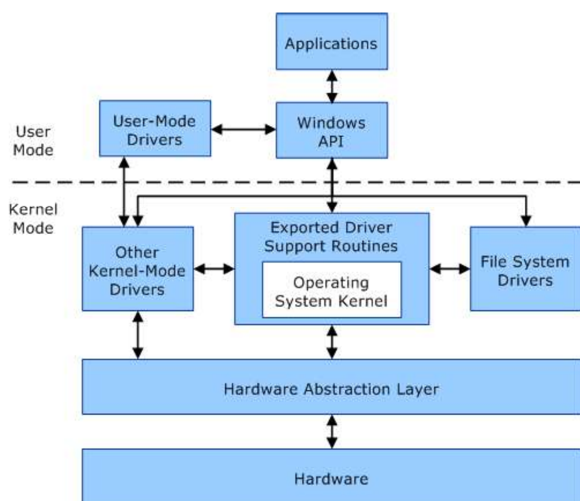

Fig. 1. Windows Syscall Orchestration

User Mode:
At the top level, user applications initiate system calls to request services from the operating system, such as file operations, memory allocation, or process management. The Windows API serves as the interface between user-mode applications/drivers and the kernel. It provides a set of functions that applications can call to request services from the operating system. User-Mode Drivers operate in user space and interact with the Windows API to facilitate communication between applications and the kernel. They handle tasks that do not require direct hardware access.

Kernel Mode:
This is the core of the operating system where the actual system calls are processed. It has higher privileges and direct access to hardware and system resources. The kernel is the central component of the operating system, managing system resources, hardware communication, and ensuring that applications and drivers operate correctly and securely. Specialized kernel-mode drivers manage file system operations, such as reading and writing data to storage devices. Other Kernel-Mode drivers operate within the kernel and provide additional functionality, such as network communication or device management, by interacting directly with the hardware or other kernel components. Kernel-mode drivers can use Exported Driver Support routines provided by the kernel to perform their functions. These routines act as intermediaries between the drivers and the core kernel. The Hardware Abstraction Layer (HAL) provides a uniform interface to the hardware, abstracting the differences in hardware architecture. This allows the kernel and drivers to interact with hardware without needing to know the specific details of the hardware. At the bottom layer, the physical hardware components (CPU, memory, storage, etc.) execute the instructions and store the data as managed by the layers above.

III. Literature Review

This survey examines the research landscape of in-memory and fileless malware execution techniques, focusing on their efficacy in evading detection. These techniques pose a significant challenge to traditional security solutions, as they minimize or eliminate file-based artifacts, making signature-based detection difficult.

A. Memory Resident Techniques:

Research emphasizes the increasing use of LOLBins, legitimate system tools like PowerShell and WMI, to execute malicious code. Elghaly (Elghaly, 2024) describes how PowerShell's ability to execute scripts directly in memory makes it a powerful tool for fileless attacks. This aligns with the trend of attackers leveraging trusted system components, as highlighted in Khushali (Khushali, 2020). The overlap with and utilization of "living off the land" techniques is a significant area of ongoing research.

C. Detection and Mitigation Strategies:

While traditional AV solutions struggle with fileless malware, research explores alternative approaches:

- a. Memory Scanning: Examining running processes for suspicious code or patterns.
- b. Behavioral Analysis: Monitoring for anomalous process behavior. Sudhakar and Kumar (Sudhakar & Kumar, 2020) suggest focusing on processes with elevated privileges and irregular system calls as potential indicators of fileless activity.
- c. API Hooking Detection: Flagging suspicious attempts to intercept system calls (Rodchenko, 2023) focuses on these activities in the CLR, a common target for manipulation.
- d. Memory Forensics: Analyzing memory dumps for evidence of in-memory execution.

IV. Understanding Edr Api Hooking

EDR, or Endpoint Detection and Response, monitors endpoint devices like laptops and servers for malicious activity. It goes beyond traditional antivirus by focusing on behavioral analysis rather than just signature matching.

TABLE I.

	Problem Statement	Existing Systems	Limitations	Outline Approach
1	Traditional security struggles with fileless malware, lacking file-based artifacts for detection.	Traditional Antivirus, Endpoint Detection and Response	AV uses file signatures, bypassed by fileless execution. EDR may lack visibility into memory-based activities.	Explore detection and mitigation strategies, including behavioral analysis and memory forensics.
2	Fileless malware evades traditional security by operating in memory, posing a significant threat.	Traditional Antivirus, Static and Dynamic Analysis Tools	Traditional AV fails against memory-resident payloads; static/dynamic analysis can be evaded by advanced techniques.	Exploit vulnerabilities, use LOLBins for propagation. Counter with memory scanning, behavioral analysis.
3	The increasing prevalence of fileless malware presents a significant challenge to incident response and traditional security approaches.	Antivirus, Sandboxes, Behavior-Based Heuristics, Unsupervised Machine Learning	Fileless malware evades detection by operating in memory, misusing legitimate tools, and displaying subtle behaviors, bypassing file based defenses	Examine common EDR solutions and their ability to detect in-memory code execution bypassing API hooks.
4	Existing PDF malware scanners are easily evaded by increasingly sophisticated obfuscation and evasion techniques.	Static and Dynamic PDF Malware Scanners	Scanners falter against complex, combined evasions, whose side effects further hinder detection.	Proposes a methodology for testing the anti-evasion capabilities of PDF malware scanners.

A. Hooked EDR – Execution Flow

EDR vendors rely on techniques like user-land hooks to detect malicious activities as Kernel Patch Protection prevents them from interacting directly with Kernel operations [12]. The typical process for EDR vendors to inject hooks into user-land processes is as follows:

1) The EDR service or driver intercepts the call to start a user process.
2) The EDR service maps its user-land hook detection DLL into the process's memory space.
3) The EDR service injects assembly instructions (usually `JMP` instructions) into each Windows API function it wishes to intercept. These instructions redirect execution to a memory location within the EDR's detection DLL.
4) The user process starts normally after the hooks are injected.

Figure 1 illustrates this process at a high level, showing how hooks and a DLL are injected into an attacker-controlled process. The goal of this defense mechanism is to intercept any Windows API call that could be used for malicious purposes, such as reading/writing memory in specific regions or connecting to malicious IP addresses. The EDR solution inspects the function call's parameters and decides whether to block or allow the call.
It is important to note that applications typically call Windows API functions (e.g., `ReadProcessMemory`, `WriteProcessMemory`) by referencing the exported function address in `kernel32.dll`. This address forwards the call to a similar assembly code stub in `kernelbase.dll`, which then forwards the call to a low-level NTAPI function in `ntdll.dll` (e.g., `ZwProtectVirtualMemory`). Finally, a `syscall` is executed to complete the function call in kernel mode. EDR vendors usually install their intercept hooks at this final stage in `ntdll.dll`.

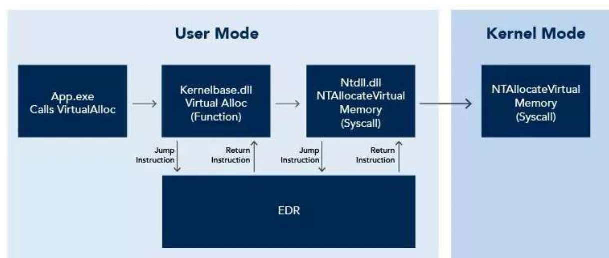

Fig. 2. Execution flow - Hooked EDR (Source) https://www.optiv.com/

B. API Unhooking Techniques

TABLE II . COMMONLY HOOKED WINDOWS API

Hooked Function	Attack Technique	Detection Limitations
NtCreateFile	Malicious File Creation	Limited set of recognized malicious file hashes or target paths
NtReadVirtualMemory	LSASS dump / credential theft	Limited set of known bad/ accessible memory region to be read
NtWriteVirtualMemory	Process Injection	Memory access /monitoring required to get alert
NtVirtualAllocEx	memory allocation in remote processes aiding in code injection.	Hard to detect without continuous integrity checks or monitoring memory allocations.

C. System Calls

EDRs (Endpoint Detection and Response systems) rely heavily on monitoring system calls to track process activity, particularly those system calls frequently exploited for malicious purposes. System calls act as the interface between a process's memory and the kernel (the operating system in memory). Any attempt to access protected resources—such as another process's memory, the file system, or the registry—must be made through a system call. The kernel enforces security policies at this level, and if access is denied, the process cannot access the requested resource unless it employs an exploitation technique like privilege escalation.

As shown by IDA and as expected, `ZwCreateFile` and `NtCreateFile` are the same function. Understanding the instructions in this function is crucial for comprehending the EDR bypass techniques discussed later. Here's a breakdown:
1)The `rcx` register is copied into `r10`. In the Windows calling convention, `rcx` holds the first parameter for function calls, while `r10` is used for the first parameter in system calls. Other registers retain their order between system and function calls.
2) The value `55h` is loaded into the `eax` register. This is the system call entry number, often referred to as the system service number (SSN) on Windows.
3) To execute a system call, the SSN must be known for the specific Windows version, as these numbers vary between versions.
4) The system then determines whether to use the `syscall` instruction or the older `int 2E` interface. Both methods ultimately lead to the kernel handling the system call.

D. EDR System Call Monitoring

EDRs monitor a wide range of system activities by hooking system call handlers within monitored processes. This allows them to intercept system calls, inspect their parameters, and deny execution if necessary. For instance, if a malicious system call is detected—such as creating a thread in a writable and executable (RWX) memory region—the EDR can block the call and prevent the malware from deploying its payload. This not only halts the malware but also enables the EDR to generate an alert detailing the malicious activity.
Malware authors aim to evade such detection while still utilizing the system calls required to achieve their objectives. To this end, modern malware employs various evasion techniques designed to disrupt an EDR's ability to monitor system calls.

V. Evasion Techniques

We will be comparing 3 ways to evade detection by EDR.

A. Overwriting NTDLL with clean version / Model Unhooking

Background
For most of the syscalls NTDLL is the last library in user mode, inside NTDLL it passes execution to kernel mode. Common Api calls go through NTDLL, e.g. NtAllocateVirtualMemory calls VirtualAlloc in kernel32. EDR hooks into NTDLL to analyze and correlate the data. To unhook EDR original ntdll.dll copy is obtained from disk (unhooked dll). This copy is overwritten in own process memory to bypass EDR hooking. Drawbacks – EDR can flag accessing ntdll.dll from disk, as it is a common way of EDR unhooking process. The API calls to overwrite ntdll.dll might reside in hooked ntdll.dll. The first technique we explore is module unhooking, also referred to as API unhooking or system call unhooking. The primary objective of this technique is to evade EDR detection by restoring system calls to their original state, effectively removing the hooks that EDRs use to monitor for malicious activity.
To hook a system call, EDRs locate it within a target process and overwrite its implementation (instructions) to gain control when the system call is invoked. This creates a discrepancy between the original instructions of the function on disk and the modified instructions written by the EDR. By unhooking a function, malware rewrites the function's implementation to its original state, eliminating the EDR's ability to monitor its usage. This allows the malware to use the unhooked APIs without fear of detection through userland system call monitoring.

Internals

Malware employs two primary techniques to unhook functions. For explanation, we will use `ntdll.dll` as our example DLL, as it is the primary handler for system calls and a common target for unhooking by malware.
Overwriting the Entire .text Section:
The first technique leverages the fact that all system call handling code in `ntdll.dll` resides in its `.text` section. Consequently, any EDR hooks are also written within this section. To unhook all APIs simultaneously, malware first obtains a clean, unhooked copy of `ntdll.dll`. It then parses the metadata of this clean copy to locate the `.text` section and reads its contents. This clean `.text` section is then

used to overwrite the corresponding section in the active `ntdll.dll` within the process, effectively removing all EDR hooks at once.

Selective Overwriting of System Call Handlers:

The second technique avoids overwriting the entire `.text` section and instead targets only the system call handlers either all of them or just those required by the malware. The outcome is the same: the EDR hooks are replaced with the original, unhooked versions of the system call handlers.

B. Direct System Calls

Background

It includes implementing own syscalls in assembly. This way, ntdll is bypassed which is used for kernel syscalls otherwise. SSN (System Service Number) plays an important role in this technique. It identifies which syscall executes. The syscall number change not only between Windows OS versions (XP, 7, 8, 10) but also frequently between service pack levels and patch levels. It can be obtained dynamically. It is conveniently automated by SysWhispers2. SysWhispers2 does a lot of process automation for us. It does give us all the details needed to pack our own direct system calls in our assembly.

The evasion techniques discussed earlier primarily focused on module unhooking, which allows malware to execute system calls without detection. While effective, unhooking can be extreme, especially when overwriting the entire `.text` section, and in some cases, it may lead to system instability. Additionally, EDRs can detect the removal of their hooks, flagging malicious activity within a process or group of processes.

To address these challenges, alternative bypass methods have been developed. These methods enable undetected system call usage while leaving EDR hooks intact (i.e., no overwriting). One such method is Direct System Calls. In this approach, malware retrieves the System Service Number (SSN) of a desired system call and directly invokes the `syscall` instruction, bypassing the hooked system call handler in `ntdll.dll`. This technique has been widely adopted.

Internals

One of the earliest documented implementations of direct system calls was by Cornelis [16], who released an open-source project called Dumpert. This project demonstrated the technique but had a significant limitation: it relied on hardcoded system call numbers during compilation. Since System Service Numbers (SSNs) change with each Windows build and lack a guaranteed order, this approach was impractical for real-world use.

To overcome this limitation, the Hell's Gate technique [17] was developed. Hell's Gate dynamically retrieves SSNs by parsing the export table of `ntdll.dll` to locate the `Nt*` function for the desired system call. It then analyzes the function's instructions to extract the corresponding SSN. Once the SSN is obtained, the `syscall` instruction can be executed directly after setting up the necessary parameters.

Fig 3 shows the HellsDescent function from Hell's Gate, which is responsible for invoking the `syscall` instruction. The `wSyscall` variable holds the SSN of the system call to be executed. However, Hell's Gate has a limitation: it requires access to a pristine (unhooked) copy of `ntdll.dll` to accurately determine SSNs. Analyzing a hooked copy of `ntdll.dll` would yield inconsistent results across different EDR vendors. As discussed earlier, attempts to access a clean copy of `ntdll.dll` can be detected by EDRs or through advanced memory forensics techniques.

This limitation led to the development of several variations of Hell's Gate, such as Halo's Gate [18], Tartarus Gate [19] and few others. These techniques allow for retrieving SSNs without requiring access to an unhooked `ntdll.dll`. Additionally, the SysWhispers2 project [21] automates address sorting, simplifying the development of code that utilizes direct system calls.

Key Points:

- 1. Direct System Calls bypass EDR hooks by directly invoking the `syscall` instruction.
- 2. Hell's Gate dynamically retrieves SSNs by parsing `ntdll.dll` but requires an unhooked copy.
- 3. Advanced techniques like Halo's Gate and Tartarus Gate eliminate the need for a pristine `ntdll.dll`.
- 4. SysWhispers2 automates SSN retrieval, making direct system calls more accessible for developers.

```
HellsDescent:
    ; Move the SSN (System Service Number) into RAX
    mov eax, wSyscall  ; wSyscall holds the SSN for the desired syscall

    ; Move parameters into the correct registers (RCX, RDX, R8, R9)
    mov r10, rcx       ; Move first parameter (RCX) to R10 (volatile register)
    mov rcx, rdx       ; Move second parameter (RDX) to RCX
    mov rdx, r8        ; Move third parameter (R8) to RDX
    mov r8, r9         ; Move fourth parameter (R9) to R8

    ; Execute the syscall instruction
    syscall            ; Invoke the syscall

    ; Return from the function
    ret
```

Fig. 3. Implementation of HellsDescent

C. Indirect System Calls

Instead of directly calling syscall like direct syscall method, one can find syscall instruction in ntdll.dll and jump to that location. Techniques like "Hell's Gate" or "Halo's Gate" are used to resolve and execute syscalls dynamically.

Background

As discussed earlier, EDRs initially countered direct system calls by detecting syscall invocations originating from modules outside the expected DLLs (e.g., `ntdll.dll`). To bypass this detection, the indirect system calls technique was developed. This method works by locating a `syscall; ret` instruction pair within `ntdll.dll` and redirecting control flow to it using a `jmp` instruction. Instead of the malware directly invoking the `syscall` instruction, the call appears to originate from within `ntdll.dll`, making it harder for EDRs to detect. This approach ensures that the callstack shows `ntdll.dll` as the source of the syscall, bypassing EDRs that only verify the module responsible for the invocation.

Internals

The implementation of indirect system calls is demonstrated in the HellHall project [22], as shown in Figure 4. The technique involves using a `jmp` instruction to redirect execution to the `syscall` instruction within `ntdll.dll`. This partially obscures the origin of the syscall, as the callstack will point to `ntdll.dll` instead of the malicious code. However, the malware must still properly set up the `EAX/RAX` and `R10` registers, like direct system calls, to ensure the syscall executes correctly.

Key Points:

- 1. Indirect System Calls bypass EDR detection by making syscalls appear to originate from `ntdll.dll`.
- 2. Technique: A `jmp` instruction redirects control flow to a `syscall; ret` pair within `ntdll.dll`.
- 3. Callstack Obfuscation: The callstack shows `ntdll.dll` as the source, evading EDRs that rely on module verification.
- 4. Register Setup: The malware must still configure `EAX/RAX` (syscall number) and `R10` (first parameter) correctly.This paraphrased version maintains the technical details while improving clarity and flow. Let me know if you need further refinement!

VI. Results

This survey paper examines existing research on the efficacy of in-memory and fileless execution techniques for evading detection. Our analysis focuses on how these techniques exploit system vulnerabilities and utilize legitimate tools to achieve both persistence and evasion. The survey reveals that current security solutions, such as traditional antivirus and endpoint detection and response, face significant challenges in detecting and mitigating these attacks.

```
; Assume:
; - syscallStub: Address of the syscall instruction in ntdll.dll (e.g., syscall; ret)
; - syscallNumber: System Service Number (SSN) for the desired syscall
; - param1, param2, param3, param4: Parameters for the syscall

hellhall_indirect_syscall:
    ; Load the syscall number into RAX
    mov eax, syscallNumber  ; SSN for the desired syscall (e.g., NtAllocateVirtualMemory)

    ; Move parameters into the correct registers (RCX, RDX, R8, R9)
    mov r10, rcx            ; Move first parameter (RCX) to R10 (volatile register)
    mov rcx, param1         ; Load first parameter into RCX
    mov rdx, param2         ; Load second parameter into RDX
    mov r8, param3          ; Load third parameter into R8
    mov r9, param4          ; Load fourth parameter into R9

    ; Jump to the syscall stub in ntdll.dll
    jmp qword ptr [syscallStub]  ; Redirect execution to the syscall instruction
```

Fig. 4. Implementation of HellHall Indirect syscall

Several key trends emerge from the surveyed research:

1) Increasing Sophistication of Evasion Techniques: Attackers are continuously developing new and more sophisticated methods to bypass security solutions. This includes advanced obfuscation techniques, LOLBins exploitation, syscall manipulation and many others.

2) Limitations of Current Security Solutions: Traditional AV and EDR solutions struggle to detect and mitigate fileless malware due to their memory-resident nature and reliance on legitimate tools. Memory scanning and behavioral analysis techniques show promise but face challenges in terms of performance overhead and accuracy.

3) The Need for a Multi-Layered Approach: Effectively combating fileless malware requires a multi-layered security approach that combines traditional signature-based detection with advanced behavioral analysis, memory forensics, and threat intelligence.

This survey provides a comprehensive overview of the current state of research on fileless malware and its evasion techniques, highlighting the challenges and opportunities for improving security defenses. It sets the stage for future research by identifying critical areas for further investigation, such as developing more effective detection methods and enhancing existing security solutions to counter these evolving threats.

VII. Conclusion

In-memory and fileless execution techniques represent a significant challenge to modern security defenses. Our survey confirms that these techniques are highly effective in evading both traditional signature-based antivirus and more advanced EDR solutions. The use of indirect syscalls further complicates detection and mitigation efforts.

While memory scanning, behavioral analysis, and API hooking detection offer some countermeasures, the dynamic nature of these attacks necessitates continuous improvement in security tools and methodologies. Defenders must adopt a multi-layered approach that combines static and dynamic analysis, memory forensics, and threat intelligence to effectively combat these evolving threats. Further research is crucial to understand the limitations of current security solutions and to develop more robust and proactive defense mechanisms. Areas for future work include investigating novel evasion techniques, exploring advanced detection methods that incorporate machine learning and AI, and developing mitigation strategies that minimize the impact of these attacks.

Fileless malware and "Living-off-the-Land Binaries" represent a significant escalation in cyber threats, turning legitimate tools and system components against us. By understanding the tactics employed by attackers, using trusted processes and residing in memory and proactively bolstering our defenses, we can mitigate these risks. Vigilance and preparedness are paramount in protecting our digital infrastructure in this era of increasingly sophisticated and stealthy attacks.

VIII. Future Scope

As attack vectors and defenses both are forwarding with fast pace, the future of defending security products seems exciting and hopeful.
Combination of Memory forensics and machine learning is one probable solution. Another such technology is ATMD. Automated Moving Target Defense (AMTD) technology protects against threats by dynamically and randomly altering the runtime memory environment, creating a moving target for attackers. Decoy traps are left in the original memory locations. Legitimate applications are aware of these changes, but malicious code attempting to interact with the decoys triggers termination of the compromised process and captures forensic data.

This proactive, deterministic approach effectively stops fileless attacks, supply chain attacks, and ransomware by disrupting the attacker's assumptions about the memory layout. Soon we can focus on exploring and testing effectiveness of ATMD and more such techniques.

References

[1] R. Dewan and V. Sivakumar, "A Deep Dive into Detecting and Investigating Fileless Malware," 2023.
[2] Y. M. Elghaly, "Stealth in Plain Sight: The Hidden Threat of PowerShell Fileless Malware and Its Evasion of Modern EDRs & AVs," 2024. M. V. Kaware and A. Butalia, "Detection of Smart Android Malware Employing Deep Learning and Machine Learning," International Journal of Engineering and Research Technology (IJERT), vol. 11, no. 8, pp. 212–217, Aug. 2022, doi: 10.17577/IJERTV11IS080107.
[3] M. V. Kaware and A. Butalia, "Detection of Smart Android Malware Employing Deep Learning and Machine Learning," International
[4] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in Proc. 8th ACM Conf. Comput. Commun. Secur. (CCS), Philadelphia, PA, USA, 2001, pp. 255–264.
[5] A. Alice, "EDR Bypass: Retrieving Syscall ID with Hell's Gate, Halo's Gate, FreshyCalls and Syswhispers2," 2023. [Online]. Available: https://alice.cli.rs/edr-evasion-syscalls/
[6] Entropy-z, "EDR_Evasion_101: Ways to evade EDR hooking using ntdll unhooking and direct syscall.," 2022. [Online]. Available: https://github.com/Entropy-z/EDR_Evasion_101
[7] D. Feichter, "Direct Syscalls vs Indirect Syscalls," 2023. [Online]. Available: https://medium.com/@danielfeichter/direct-syscalls-vs-indirect-syscalls-29a961a51902
[8] V. Khushali, "A Review on Fileless Malware Analysis Techniques," J. Malware Res., vol. 8, no. 3, pp. 112–125, 2022.
[9] M. Sikorski and A. Honig, Practical Malware Analysis: A Hands-On Guide to Dissecting Malicious Software. San Francisco, CA, USA: No Starch Press, 2012.
[10] S. Sudhakar and S. Kumar, "An Emerging Threat Fileless Malware: A Survey and Research Challenges," Cybersecurity, vol. 3, no. 1, 2020.Z. Stein. (Oct. 2020). Blinding EDR on Windows. [Online]. Available: https://synzack.github.io/Blinding-EDR-On-Windows/

[11] A. Butalia, D. Bhattarcharya, and K. Satpute, "Probabilistic Integration Random Forest Decision Tree Fusion Model," International Research Journal of Engineering and Technology (IRJET), vol. 11, no. 3, Mar. 2024, ISSN: 2395-0056.

[12] "A practical guide to bypassing userland api hooking," https: //www.advania.co.uk/insights/blog/a-practical-guide-to-bypassing-u serland-api-hooking/, 2022.

[13] A Practical Guide to Bypassing Userland API Hooking. (2022).[Online]. Available: https://www.advania.co.uk/insights/blog/a-practical-guide-to-bypassing-userland-api-hooking/

[14] APT Group Chimera. (2022). [Online]. Available: https://cycraft.com/download/CyCraft-Whitepaper-Chimera-V4.1.pdf

[15] Hell'sGate. (2020). [Online].Available: https://github.com/am0nsec/HellsGate/blob/master/hells-gate.pdf

[16] A Practical Guide to Bypassing Userland API Hooking. (2022).[Online]. Available: https://www.advania.co.uk/insights/blog/a-practical-guide-to-bypassing-userland-api-hooking/

[17] Red Team Tactics: Combining Direct System Calls and sRDI to Bypass AV/EDR. (2019). [Online]. Available: https://www.outflank.nl/blog/2019/06/19/red-team-tactics-combining-direct-system-calls-and-srdi-to-bypass-av-edr/

[18] "Halo's Gate," https://blog.sektor7.net/#!res/2021/halosgate.md, 2021.

[19] "Tartarus Gate," https://trickster0.github.io/posts/Halo's-Gate-Evolves-t o-Tartarus-Gate/, 2021

[20] Bypassing User-Mode Hooks and Direct Invocation of System Calls for Red Teams. (2020). [Online]. Available: https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/

[21] SysWhispers2. (2022). [Online].Available: https://github.com/jthuraisamy/SysWhispers2]

[22] HellHall. (2023). [Online]. Available: https://github.com/Maldev-Academy/HellHall

[23] O. Khalid, S. Ullah, T. Ahmad, M. H. Yousaf, and K. H. Kim, "An Insight into the Machine-Learning-Based Fileless Malware Detection," Sensors, vol. 23, no. 12, p. 612, 2023.