Evading Windows Defender Antivirus With a Custom Sliver Stager

Charles T. Zhao

Project Report submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Engineering
in
Computer Engineering

Scott Midkiff, PhD, Chair
Randy Marchany
Haining Wang, PhD

May 15, 2025
Blacksburg, Virginia

Abstract

As security products become more sophisticated, attackers and red teams must similarly adapt their techniques to maintain post-exploitation access without detection. Understanding how antivirus software can be bypassed helps improve both offensive tooling and defensive countermeasures. This report explores how a custom-built C++ stager can be used to deliver and execute a Sliver Command and Control (C2) implant while evading Microsoft Defender Antivirus. By employing in-memory execution, self-injection, dynamic API resolution, and obfuscation, the stager successfully bypassed Defender's detection mechanisms in a controlled virtual environment. Results show that while traditional remote process injection techniques were flagged, the self-injecting stager maintained an active Sliver session without interruption. The project highlights both the effectiveness and limitations of such evasion methods and underscores the need for deeper behavioral and memory analysis in modern defensive tools.

1 Introduction

In cybersecurity breaches, attackers often seek not only to compromise a system, but also to establish a method of returning to it later. To accomplish this, they deploy persistent mechanisms that allow them to reconnect and control the compromised machine even after initial entry. Command and Control (C2) frameworks are commonly used in offensive cybersecurity to establish persistent remote access to a target machine. Once a system is compromised, an attacker sets up a server and deploys an implant, a payload that connects back to the server and allows the attacker to connect to the target system and remotely execute commands. For the implant to be effective, it must first be delivered and executed without being detected or blocked by antivirus software. This challenge makes evasion a critical part of the attack process. In this report, we explore how to deliver and activate a Sliver implant onto a Windows machine while avoiding detection by Microsoft Defender Antivirus. To accomplish this, we develop and deploy a custom implant stager, which is a lightweight program designed to download, decrypt, and execute the full implant entirely in memory. In doing so, we highlight both strengths and blind spots of Defender Antivirus, offering practical insight into which evasion techniques are effective and why. This work contributes educational value by revealing how evasion strategies are adapted to bypass modern security tools, which defensive mechanisms are most resilient, and where improvements are still needed. The complete source code and configuration scripts used in this work can be found at https://github.com/ChamberZ1/sliver/tree/master.

1.1 Objective

As tools and techniques in both offensive and defensive security evolve rapidly, many existing resources and walkthroughs become outdated or incompatible with current systems. While previous studies have demonstrated various techniques for evading antivirus solutions, most were conducted at least a year prior to this work. Given the continuous evolution of Microsoft Defender, this project evaluates whether those techniques remain effective in 2025 by implementing a custom Sliver stager designed to evade detection under current Microsoft Defender Antivirus capabilities. Microsoft Defender Antivirus is deployed by default across millions of Windows systems, making it one of the most widespread and critical security controls in modern computing environments. Although the original intent of this project was to explore methods for bypassing Endpoint Detection and Response (EDR) systems—a more advanced class of security solutions—the scope was intentionally narrowed to focus first on Microsoft Defender. By designing, implementing, and testing a custom C++ stager capable of in-memory Sliver implant delivery without triggering detection, this work offers practical insight into offensive evasion strategies and the limitations of Defender's detection mechanisms.

2 Background

2.1 Command and Control Frameworks

Command and Control (C2) frameworks are a critical component of modern offensive cybersecurity operations, often leveraged both by malicious actors to maintain control over compromised systems and by security professionals during penetration testing and red team exercises to simulate realistic attack

scenarios. These frameworks enable an attacker to have remote access to one or more compromised target machines while maintaining stealth and flexibility. A C2 framework is typically organized in a client-server model, as illustrated in Figure 1, where implants deployed on victim machines connect back to the C2 server to receive and execute commands[1].

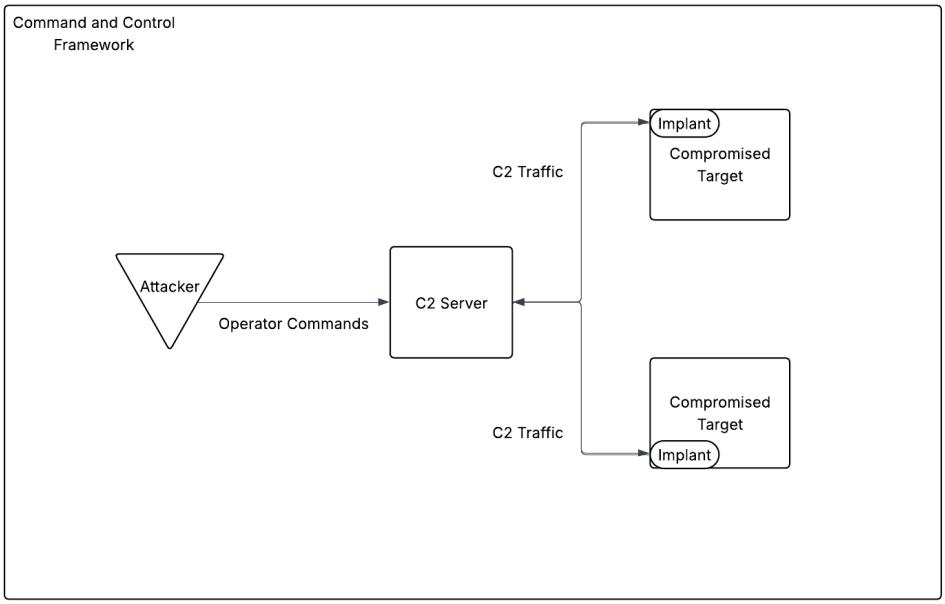

Figure 1: Overview of a C2 framework

2.2 Implants

An implant is a piece of malware deployed onto a target machine after initial compromise. Its purpose is to establish and maintain communication with the attacker's C2 server, and serve as an instrument through which the attacker can execute commands on the target system [1]. Ideally, implants are small and lightweight, minimizing their footprint on the system to avoid detection. However, Sliver implants are written in Golang [2] to support cross-platform compatibility [3][4] and include support for multiple C2 protocols, both of which significantly increase their size. Furthermore, Sliver implants are generally designed as stage two payloads, meaning they are intended to be delivered after initial access has been established with a stager. This two-step process expects the initial stager to be small and stealthy, so that the larger, more feature-rich implant can be downloaded and executed later. As such, Sliver implants prioritize functionality over minimal size, and their developers recommend using stagers in operational contexts where file size is relevant[2]. Depending on the framework and operational needs, implants are typically configured either to establish a persistent connection (known as session implants), or to periodically "check in" with the C2 server (known as beacon implants)[2]. Regardless of setup, a key requirement is that implants remain stealthy against security measures like antivirus software. As endpoint defenses continue to evolve, raw Sliver payloads—due to their size and predictability—are increasingly flagged by antivirus and EDR systems [5]. Therefore, an evasion strategy is essential for successfully executing a Sliver implant on a protected machine.

2.3 Sliver C2 Framework

Among the available C2 frameworks, Sliver stands out for its strong capabilities and accessibility. Developed by Bishop Fox as a modern, open-source alternative [6] to Cobalt Strike—one of the most widely used commercial Command and Control platforms [1]—Sliver provides powerful post-exploitation features without the cost or restrictive licensing of Cobalt Strike [4]. Unlike Cobalt Strike's C/C++ implants, Sliver primarily uses Golang, offering improved cross-platform compatibility across Windows, Linux, and macOS systems. Among its many features, it supports dynamic payload generation, enabling users to customize implants to specific operational needs. It includes communication

over multiple protocols such as mutual TLS (mTLS), HTTP(S), and WireGuard. Built-in features like sleep masking, obfuscation, and payload encryption further enhance stealth. While antivirus evasion is not the framework's explicit focus, the developers note that "Sliver is designed to be interoperable with common techniques for bypassing anti-virus software such as packers, crypters, and stagers" [7]. Additionally, Sliver implants can be compiled with randomized build options, making each payload unique and less likely to be flagged by static signature detection. Sliver supports both session-based and beacon-based implants, and its user-friendly command-line interface (CLI), extensive documentation, and modular architecture make it easy to use and extend.

In recent years, threat actors have increasingly adopted Sliver in real-world attacks. Reports by Hunt and Hackett [8], Darktrace [9], and Cybereason [10] document its use by advanced persistent threat (APT) groups including APT29 (Cozy Bear), TA551 (Shathak), and Exotic Lily, often in conjunction with malware loaders to establish persistent access. These actors utilize Sliver's flexible evasion techniques, in-memory execution, and support for asynchronous (beacon) and real-time (session) communication modes. Its modular architecture and the Armory extension system further enhance operational flexibility, allowing threat actors to integrate third-party post-exploitation tools such as Beacon Object Files (BOFs), Rubeus, Seatbelt, and Mimikatz directly into their workflows. Sliver's growing popularity among adversaries and professionals is often attributed to its obscurity compared to more well-known frameworks like Cobalt Strike as well as its technical strengths, accessibility, and flexibility. The fact that Sliver is used not only for adversary emulation but also by real-world threat actors underscores its practicality, effectiveness, and growing prominence as a C2 framework.

2.4 Stagers

Stagers play an important role in the stealthy deployment of payloads, particularly in environments with strict security controls. A stager is a small initial payload that connects to an attacker-controlled server to retrieve and execute the actual payload. Because of stagers, the beginning infection vector has a smaller footprint, making it less likely to trigger security alerts compared to directly delivering a full implant, which is especially helpful in Sliver's case, as its implants are known to be large[7][2].

2.5 Microsoft Defender Antivirus

Due to Windows being such a popular operating system, Microsoft Defender Antivirus represents one of the most common security controls encountered during offensive security operations. Defender is installed by default on Windows operating systems and has evolved significantly over the years. Most notably, on top of traditional signature-based detection, modern iterations incorporate machine learning (ML) models, behavior analysis, cloud-based protection, and regular signature updates [11]. With its widespread deployment across enterprise and personal environments, successfully evading Microsoft Defender has become a fundamental requirement for offensive security operations. Understanding Defender's detection mechanisms and what detection technique they are classified as is essential for developing effective evasion techniques. Table 1 shows a mapping of Defender components to detection techniques.

Defender Antivirus uses a multi-layered, hybrid approach to threat detection and prevention, combining lightweight client-side engines with more powerful cloud-based analysis [12].

2.5.1 Metadata-Based ML Engine

To start, the local ML models analyze specific file types commonly exploited by attackers, providing a verdict within milliseconds. If a file is flagged as suspicious or uncertain, it may be escalated to the cloud, where metadata-based engines perform deeper analysis by combining the outputs of multiple specialized models.

2.5.2 Behavior-Based ML Engine

Secondly, Defender monitors process behaviors at runtime to detect potential attacks. On the client side, the behavior monitoring engine observes post-execution activity and identifies suspicious behavior based on given rules. If suspicious behavior is detected, it may be escalated to cloud-based machine learning models for more thorough analysis.

2.5.3 Antimalware Scan Interface (AMSI)-Paired ML Engine

Next, Defender monitors both process memory and dynamic code execution to expose hidden malicious behavior. The memory scanning engine inspects the memory space of running processes to uncover threats such as obfuscated code. To address script-based attacks, Defender leverages the Antimalware Scan Interface (AMSI), a standard interface that allows applications to pass decoded script content to antivirus engines for real-time inspection. Through AMSI integration, Defender can detect and block obfuscated or malicious scripts either before or during execution on the client side. For deeper analysis, suspicious files or scripts may be escalated to the cloud, where full file contents are classified using deep neural network models. Within seconds, these models provide a verdict for the file.

2.5.4 Detonation-Based ML Engine

Heuristics rules on the client are used to identify characteristics associated with known malware families or behaviors. In the cloud, detonation-based engines execute suspicious files in sandbox environments to dynamically observe behaviors, enabling detection of sophisticated malware that may attempt to evade static scanning.

2.5.5 Reputation ML Engine

Emulation engines further support local defenses by dynamically unpacking potentially malicious content and scanning runtime behavior, revealing polymorphic or packed malware. Cloud-side reputation models, informed by various Microsoft sources, further assist in blocking threats linked to known malicious or suspicious Uniform Resource Locators (URLs), domains, emails, and files.

2.5.6 Network and Smart Rules Engine

Defender also incorporates network activity monitoring on the client to detect malicious communications, and smart rules engines in the cloud leverage expert knowledge to identify emerging threats.

2.5.7 CommandLine ML Engine

Finally, command-line scanning engines inspect process command lines locally before execution, while cloud-based command-line models evaluate suspicious patterns and remotely instruct the client to block malicious processes.

This layered integration of client and cloud technologies enables Microsoft Defender to provide rapid, highly accurate threat detection and response, allowing it to block emerging malware techniques and sophisticated attack chains.

Table 1: Mapping of Microsoft Defender Antivirus Components to Detection Techniques

Defender Component	**Detection Technique Represented**
Metadata-Based ML Engine	Static analysis (enhanced with metadata and machine learning)
Behavior-Based ML Engine	Behavioral analysis (runtime behavior monitoring)
AMSI-Paired ML Engine	Memory inspection and script-based behavioral analysis
Detonation-Based ML Engine	Heuristic analysis via sandboxing and behavioral execution tracing
Reputation ML Engine	Signature-based and heuristic detection using known threat intelligence
Network and Smart Rules Engine	Network-based behavioral analysis and rule-based threat detection
CommandLine ML Engine	Static and behavioral inspection of command-line activity

2.6 In-Memory Execution

In-memory execution, also known as fileless execution, is a technique in which malicious code is executed directly in memory, bypassing the need to write files to disk. This method has become increasingly popular due to its effectiveness in evading traditional static detection mechanisms.

Unlike conventional malware, which drops executables to disk where they can be scanned before execution by antivirus (AV) or Endpoint Detection and Response (EDR) tools, fileless malware operates entirely in random access memory (RAM) and leaves no files behind. This makes it extremely difficult to detect using static analysis, signature-based detection, or even some behavioral tools. Products like Microsoft Defender typically scan files using hash comparisons, import address table (IAT) analysis, or YARA rules [13], which look for known strings, byte patterns, or file structures. But when no file exists on disk, when no file is available to be analyzed, these defenses become ineffective.

Once a payload is injected into memory, it leaves no static artifacts, and detection must rely on behavioral analysis and runtime memory inspection. Techniques used to achieve in-memory execution include remote thread injection, process hollowing, and local shellcode injection. These methods allow attackers to load and run malicious code entirely within the memory space of a process, often hiding inside legitimate system processes to further reduce visibility [14].

Modern fileless attacks also use obfuscation, encryption, packing, or custom loaders that decrypt or unpack code only at runtime, making analysis even harder. According to a Ponemon Institute study [15] and report by IBM [16], fileless malware is up to ten times more likely to succeed than traditional file-based attacks. As a result, in-memory execution remains a key strategy in modern offensive operations—particularly when bypassing AV and EDR is essential.

2.7 Process Injection

Process injection[17] is a common technique used to execute arbitrary code within the context of another process. This can either be a separate, trusted process (remote injection) or the originating process itself (self-injection).

Remote Process Injection involves opening a handle (a unique identifier used by the operating system to reference a resource like a process, file, thread, or DLL) to another running process (e.g., `notepad.exe`), allocating memory in that process using `VirtualAllocEx`, writing shellcode with `WriteProcessMemory`, and executing it via `CreateRemoteThread`. This method is commonly flagged by Defender due to its behavioral signatures. Cross-process memory operations and remote thread creation are highly suspicious and often associated with malware[18].

Self-Injection, on the other hand, involves allocating memory and executing code within the current process. This avoids API function calls on another process, reducing the behavioral red flags. Instead, a thread is created internally using standard calls, and no external processes are touched. While not entirely undetectable, self-injection has a lower risk profile under typical heuristic rules.

Although remote injection allows us to hide the execution of shellcode within a trusted Windows process, self-injection is particularly useful in environments where EDR or AV solutions closely monitor process creation and injection into trusted binaries. Combined with obfuscation and dynamic API resolution, self-injection can be a significantly stealthier alternative to remote injection techniques.

2.8 Dynamic API Resolution

To perform process injection on Windows, three core API functions are typically required: `VirtualAllocEx`[1], `WriteProcessMemory`[2], and `CreateRemoteThread`[3]. [19, 18, 20].

These functions are typically exported by `kernel32.dll`, a core Windows system dynamic-link library (DLL) responsible for fundamental low-level operations such as process and memory management, input/output operations, and thread creation. Since `kernel32.dll` is loaded in nearly every Windows process by default, it is commonly targeted for API resolution by malware and legitimate applications alike.

However, statically linking these functions results in their names being recorded in the Import Address Table (IAT), which can be easily scanned by antivirus software such as Microsoft Defender.

[1] Allocates memory in a specified process
[2] Writes data to an allocated memory region
[3] Starts a new thread at a given memory address to execute the code

Static analysis tools often flag binaries that import combinations of API calls known to be associated with malicious behaviors like process injection.

Dynamic API resolution is an evasion technique that circumvents this limitation by resolving API function addresses at runtime rather than at compile time [21]. Instead of relying on static linkage, a program can use `GetModuleHandle`[22] to obtain a handle to `kernel32.dll` (or other system libraries) and then use `GetProcAddress`[23] to retrieve the memory addresses of the required functions dynamically. This approach eliminates the need to include sensitive APIs in the IAT, thereby avoiding detection based on static imports.

Dynamic API resolution is widely used in both red team tooling and real-world malware. By concealing operational capabilities until runtime and avoiding recognizable code patterns, it provides a practical means of evading static detection. Figure 2 shows the difference between static and dynamic API resolution.

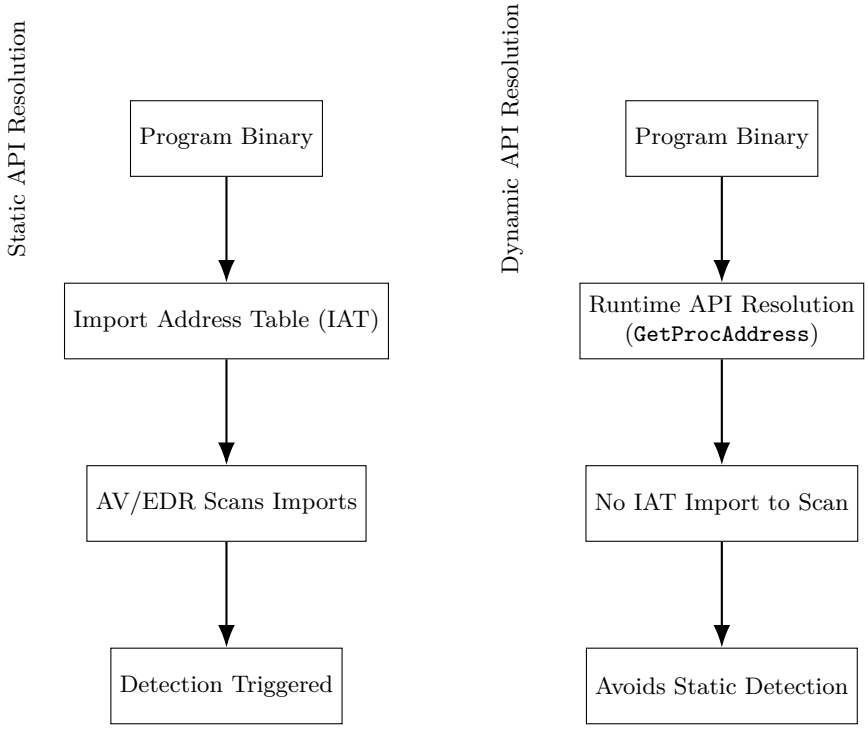

Figure 2: Comparison of Static vs. Dynamic API Resolution. Static linkage exposes sensitive API calls to signature-based scanning, while dynamic resolution hides imports until runtime.

3 Literature Review

Extensive research and practical work have long focused on bypassing antivirus defenses, particularly Microsoft Defender, which remains one of the most widely deployed protections in Windows environments. In this literature review we examine several prior studies relevant to this work.

3.1 Antivirus Evasion Techniques in C2 Frameworks

Modern Command and Control (C2) frameworks such as Sliver incorporate numerous features to evade local antivirus defenses like Microsoft Defender. Hummel [24] outlines several of these tactics, emphasizing their use in post-exploitation scenarios where stealth is critical to maintaining persistent access. One key strategy is obfuscation, such as compression, encryption, or control flow tailing, which alters the logical structure of code execution to counter static analysis and detection, thereby complicating reverse engineering efforts and reducing the likelihood of Defender flagging the payload based on recognizable patterns.

In addition, encrypted payload delivery and runtime decryption further enhance stealth; in their Sliver-based demonstration, NtTeam21 [25] successfully evaded BitDefender antivirus by encrypting their payload with AES and performing in-memory decryption at runtime, thereby bypassing static detection. To further minimize visibility to defensive tools, they employed Early Bird Asynchronous Procedure Call (APC) injection and indirect syscalls. Early Bird APC injection improves stealth by queuing malicious code into a newly created, suspended thread before the target process fully begins execution, making it harder for security products to inspect or block the injection at runtime. Indirect syscalls, implemented using SysWhispers3, further enhanced evasion by bypassing user-mode hooks. User-mode hooking is a technique used by security software to intercept and monitor sensitive API calls such as `VirtualAlloc` or `CreateRemoteThread`. Instead of relying on the hooked API functions, NtTeam21's approach directly invoked system calls at the kernel level, allowing the payload to execute without triggering antivirus monitoring mechanisms. While our work does not implement advanced techniques such as Early Bird injection or indirect syscalls as demonstrated by NtTeam21 [25], their use of encrypted payloads, runtime decryption, and in-memory execution aligns closely with the evasion goals pursued here. Like our approach, their methodology focuses on minimizing disk presence and avoiding static and behavioral detection through stealthy runtime execution.

Similarly, researchers at Cybereason observed that Sliver's built-in commands such as `migrate` and `getsystem` perform process injection techniques, such as injecting into `notepad.exe` or `spoolsv.exe` using `CreateRemoteThread`, to gain elevated privileges and evade detection [10]. These actions are designed to bypass behavioral defenses by embedding malicious activity within trusted processes, making static and heuristic analysis by antivirus like Microsoft Defender more difficult.

More recently, Mishra demonstrated that minor but strategic modifications to the Sliver C2 framework can significantly improve its evasion capabilities against modern EDR systems. Mishra notes that out-of-the-box Sliver payloads are increasingly detected due to their large binary size (up to 30 MB) and static signatures embedded in protocol buffer files like sliver.proto. The researchers overcame these static detections by renaming functions and shortening messages throughout the source code (e.g., changing ScreenshotReq to ScShotReq) and propagating these changes across Sliver's auto-generated files [5]. Furthermore, changes like renaming export functions and replacing identifiable method calls such as GetJitter were automated with scripting, ensuring consistent obfuscation across builds. Testing against solutions like Elastic Agent confirmed that the customized payloads successfully bypassed both static and dynamic detections. In addition to altering static signatures, they addressed behavioral detections by disabling Sliver's default AMSI bypass which are heavily signatured, and implementing custom shellcode loaders that perform in-memory payload mapping. These adjustments reduced the risk of triggering runtime alerts. This work underscores the adaptability of open-source C2 tools like Sliver and aligns with the core evasion principles in our project—namely minimizing static indicators, performing in-memory execution, and reducing behavioral flags through lightweight and customized stagers.

Lastly, in his blog posts, Dominic Breuker's demonstration of operational use of Sliver incorporates lightweight stagers to deliver implants dynamically. Breuker reinforces the fact that Sliver implants, compiled in Golang, are inherently large and easily flagged by antivirus solutions when executed directly [26]. Instead of deploying a full implant binary, Breuker builds a small custom stager to download the implant shellcode from the C2 server and execute it in memory, minimizing the initial attack surface and avoiding file-based detection. Breuker demonstrates that process injection can significantly enhance stealth by injecting downloaded shellcode into legitimate processes like `notepad.exe` or `msedge.exe`. Using functions such as `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread`, the stager writes and executes the implant within the memory space of a trusted process, making it harder for Microsoft Defender to detect. Although the stager bypassed Defender in testing, Breuker notes that detailed logging tools like Sysmon could still expose suspicious behavior, such as remote thread creation. This highlights the importance of having layered security measures, as no single detection method is sufficient to catch all forms of evasion.

These works, among others [27][28][14], demonstrate that effective antivirus evasion relies on a combination of common malware techniques such as in-memory execution, obfuscation[29], and process injection[17]. From Hummel and Mishra's obfuscation of code, to NtTeam21's deployment of indirect syscalls and Early Bird injection, and Breuker's use of lightweight stagers and remote shellcode injection, each strategy highlights the value of minimizing on-disk traces and embedding malicious code within running processes. Drawing from these approaches, this work develops a custom stager that

executes an obfuscated Sliver implant entirely in memory, uses legitimate system APIs, and avoids leaving behind files on disk. These evasion techniques, inspired by prior research, helped us successfully avoid Microsoft Defender while keeping the implant functional.

Although Microsoft has improved its defensive countermeasures by incorporating cloud-based machine learning [30] to scale the integration of AMSI, memory scanning, and behavior monitoring for detecting runtime anomalies, research has demonstrated the limitations of relying solely on machine learning for malware detection. Studies show that adversaries can still evade classifiers through adversarial examples and structural obfuscation [31, 32, 33, 34, 35]. These evolving defensive techniques underline the need for continuously adapting offensive evasion strategies—while, conversely, the growing sophistication of such offensive techniques further reinforces the importance of improving defensive capabilities.

4 Methodology

For our approach, we developed a custom C++ stager that incorporates several evasion techniques to deliver a Sliver implant while bypassing Microsoft Defender Antivirus. This method was selected based on the recommendation of Sliver developers, as well as a thorough review of prior research, including the works of Breuker [26] and Evasive Ginger [36], which demonstrated the effectiveness of custom stagers in evading modern antivirus solutions.

4.1 Setup

Our setup consisted of two systems: a physical machine running Linux Mint 22.1, which hosted the Sliver C2 server, and a virtual machine running Windows 11 Pro version 24H2 on VirtualBox, which served as the target. Linux Mint was chosen to host the C2 server due to its Ubuntu-based stability, ease of use, and strong compatibility with penetration testing tools. In addition, the Sliver documentation recommends deploying the server on a Linux host, as certain features may be less stable or harder to configure on Windows systems [2]. Windows was selected for the target because it remains the most widely used desktop operating system, making it a realistic and high-value target in adversarial scenarios. Using a virtual machine for the target provided a controlled and flexible environment for implant delivery and testing. Virtualization allowed for rapid rollback between test iterations via snapshots, and if critical issues arose, a new instance could be easily deployed. Additionally, virtualization avoided the need for dedicated physical hardware and simplified network containment during testing.

4.1.1 Sliver

To set up the Sliver Command and Control (C2) framework, we forked the official GitHub repository maintained by Bishop Fox [37]. Forking, rather than cloning, allowed us to make custom modifications and keep our version separate from the main repository while still being able to pull in updates if needed. We followed the official documentation to build and launch the Sliver server on the Linux Mint machine.

As for choosing the implant, while beacon implants are generally considered more stealthy due to their intermittent communication and use of jitter, we opted to use session implants during testing for two key reasons.

First, session implants immediately establish a persistent connection upon execution, allowing for faster and more reliable feedback during development and validation of the implant delivery and execution process. This reduced the complexity involved in configuring and troubleshooting delayed beaconing behavior, which can silently fail if not tuned properly.

Second, the assumed stealth advantage of beacon implants is less relevant when it comes to evading Microsoft Defender Antivirus, as Defender does not perform deep inspection of outbound network traffic and lacks network-based anomaly detection capabilities that full EDR systems have. Therefore, both session and beacon implants are likely to go undetected by Defender unless flagged by static signature, behavioral analysis, or cloud-based heuristics. As such, session implants were selected to streamline testing and validation.

The following command was used to generate the implant: `generate --mtls 192.168.56.1:443 --format shellcode --name FLYING_PENCIL --save session_implant.bin`

- `generate` – Invokes the Sliver command to create a new implant.
- `--mtls 192.168.56.1:443` – Specifies that the implant will use mutual TLS (mTLS) to communicate with the C2 server at IP `192.168.56.1` on port `443`.
- `--format shellcode` – Sets the output format to raw shellcode (`.bin`), suitable for in-memory execution.
- `--name FLYING_PENCIL` – Assigns the name `FLYING_PENCIL` to the implant for session tracking.
- `--save session_implant.bin` – Saves the implant as a file named `session_implant.bin`.

Using the shellcode format instead of a standard executable (.exe) format was a deliberate choice to enhance stealth and bypass traditional static detection, as we knew we could not directly execute an implant executable on Windows while Defender was active [26]. Raw shellcode lacks the metadata and structural signatures of Windows executables (e.g., PE headers), making it less likely to be flagged during static analysis by antivirus tools like Defender. Furthermore, shellcode is ideal for in-memory execution, where it can be injected into a running process without ever touching disk—thereby minimizing the forensic footprint [20]. We chose mTLS because it offers stronger encryption for C2 traffic. Additionally, it is the default protocol recommended by the Sliver developers due to its robust security and high throughput characteristics [38].

Though AES encryption was a strong candidate for encrypting our shellcode, we opted to obfuscate the raw shellcode binary using XOR encoding as a simpler and lighter alternative, avoiding the inclusion of unnecessary cryptographic libraries and potentially risky API usage. The implementation details are available in the project GitHub repo.

To establish a session with the implant, a corresponding listener must be active on the C2 server. This was accomplished by running the `mtls` command within Sliver, which by default sets up an mTLS listener on port `443`. The `jobs` command can then be used to verify that the listener is active. Once the implant is successfully delivered and executed in memory, the `sessions` command can be used within the Sliver CLI to confirm an active session.

4.1.2 Windows Security Settings

On our target Windows machine, we have Microsoft Defender Antivirus running as the sole antivirus protection. Microsoft Defender Antivirus uses `security intelligence` to detect threats and is automatically updated. The recent security intelligence version at the time of writing is 1.427.366.0. As for `virus and protection settings` for Microsoft Defender Antivirus, initially we had all but `Automatic sample submission` on. This was due to wanting to be able to continuously test the stager against the other protection mechanisms while not risking the submission of the file each time there was a change. That would mean we would have to start from scratch each time. However, for our final tests, we had all protection settings turned on.

4.1.3 Development Environment

Following the recommendation in Breuker's blog [26] and for convenience, we opted to develop the Windows stager natively on Windows using Visual Studio 2022, allowing us to write and test the code against the target environment directly and in real time. Furthermore, the Visual Studio project folder was added as an excluded folder so that Defender would not be able to interfere during the development process.

4.1.4 Connectivity

To enable communication between the two systems, we configured a network adapter in VirtualBox using the `Host-only Adapter` option. This created a dedicated communication channel between the host and guest systems, isolated from the primary internet-facing network. While both machines retained internet access via a separate NAT adapter for tool installation and updates, the host-only

interface ensured controlled and direct communication for implant testing. To facilitate the transfer of the implant shellcode to the stager, we added a firewall rule on the Linux host to allow incoming TCP connections from the Windows IP address `192.168.56.102` on arbitrary port `8000`:

```
sudo ufw allow from 192.168.56.102 to any port 8000 proto tcp
```

Additionally, to ensure that the implant could establish a connection with the Sliver C2 listener once activated, we implemented another rule to allow incoming TCP traffic on port `443`:

```
sudo ufw allow 443/tcp
```

4.1.5 The Stager

We decided to build a custom stager as it was one of the most prevalent and effective evasion methods observed during our research. To get started, we followed the guidance outlined in Dominic Breuker's blog, which details how to construct a custom C++ stager for the Sliver C2 framework using process injection techniques [26].

Breuker's setup was a little more advanced in terms of environmental realism. His stager downloaded shellcode from a remote HTTP listener using the WinINet API and disguised the payload as a benign-looking resource (e.g., fontawesome.woff) served over port 80. This method was likely intended to blend into normal web traffic and avoid suspicion from network defenders. In contrast, our stager used a simpler approach by establishing a raw TCP connection to a listener hosted on the Linux machine. Since our operational focus was not on blending into normal web traffic, but rather on evading detection by Microsoft Defender Antivirus, the use of a simple TCP socket for transferring the implant was sufficient for our needs. Additionally, because we obfuscated the payload with XOR encoding prior to transmission and decoded it at runtime, in memory, we were confident in its invisibility to network and memory-based detection mechanisms. While Breuker's method emphasized stealth in the context of HTTP-based exfiltration, our approach prioritized bypassing static and behavioral scanning mechanisms from Defender Antivirus.

To execute the downloaded implant, Breuker injected it into a legitimate process (`notepad.exe`) using Windows API functions like `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread`. While his article suggests that he successfully demonstrated evasion against Defender Antivirus, our attempt to replicate his approach was immediately shut down by Defender. This does not come as a surprise, however, as his work is over two years old and modern AV tools have become more sophisticated. We found that statically calling these API functions easily triggered Microsoft Defender and was therefore not sufficiently stealthy for our needs.

To improve stealth and evasion, we made several key changes. First, instead of injecting into another process, we transitioned to self-injection, allocating memory within our own process at runtime and executing the shellcode internally. This eliminated the need for remote thread creation and reduced the number of suspicious API calls, making detection more difficult. Second, we added XOR-based decoding to deobfuscate the encoded shellcode payload after it has already been loaded in memory, reducing the chances of network-based or memory scanners identifying the payload. We also assigned unconventional function names (e.g., `netXfer`, `execLocal`) to reduce static readability and the likelihood of keyword-based signature detection. Furthermore, dynamic API resolution via `GetProcAddress` was employed to hide imports, avoiding telltale IAT entries that AV tools commonly scan for.

Our stager, although an executable stored on disk, is small in size and leaves a minimal footprint [39]. It fetches the actual Sliver payload and executes it in memory within its own process using dynamically resolved Windows API calls. This in-memory execution strategy avoids creating or writing suspicious files to disk and reduces the presence of indicators that static and heuristic scanners typically rely on.

While Breuker's example was invaluable in illustrating the core mechanics of a stager, we found that additional engineering was required to make it viable in environments with modern AV protections. Our method ensured that the implant remained fileless beyond the initial stager executable, running the shellcode purely in memory, and minimizing behavioral red flags.

To run the stager, we compiled the C++ code into an executable file which was output to the `Release` folder in the project directory. We then copied the executable to a non-excluded directory (`Downloads`) under a benign name, `tastethecake.exe`, as shown in Figure 3

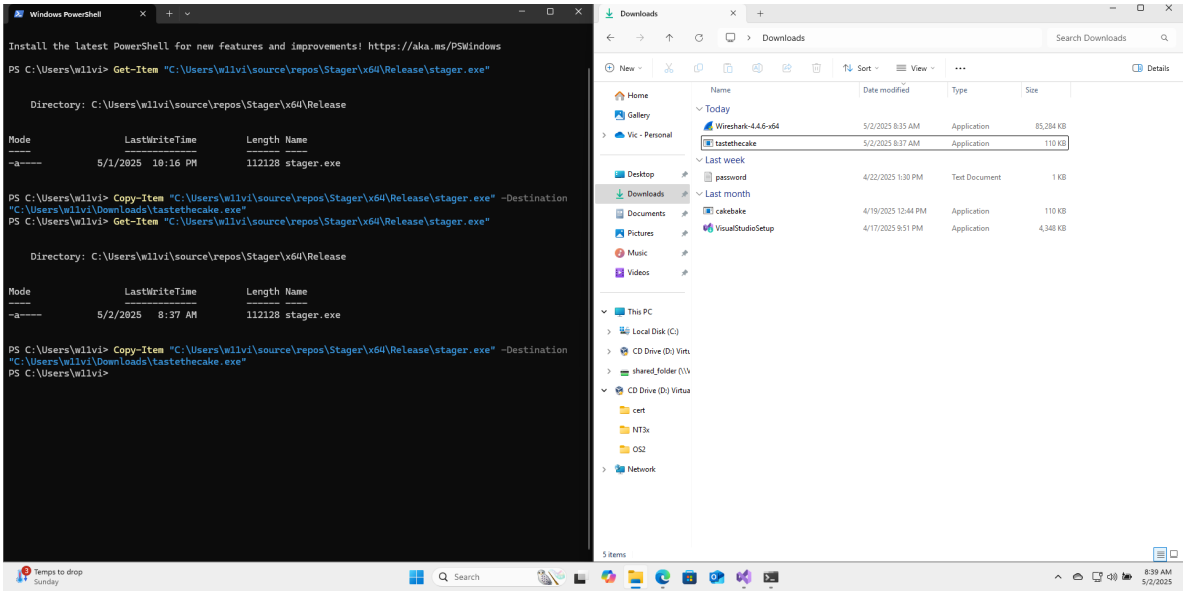

Figure 3: Powershell commands (left) used to view and copy the stager executable in the `Release` folder to a file named `tastethecake.exe` in the `Downloads` folder (right).

On the Linux machine, we prepared the listener that would serve the encoded session implant shellcode once a connection was detected, as shown in Figure 4

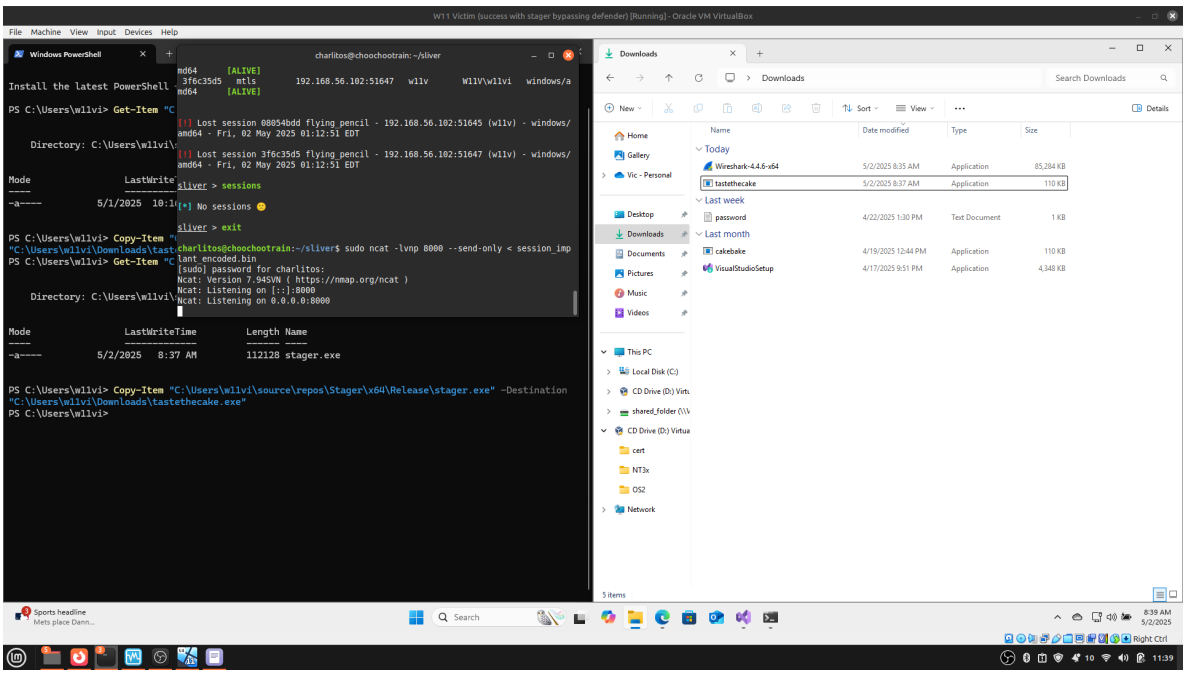

Figure 4: Active Netcat listener on the Linux machine, ready to send the encoded session implant shellcode upon receiving a connection.

To initiate the process, we simply double-clicked `tastethecake.exe`, which launched a command prompt window as a new process. Figure 5 confirms that the listener received a connection shortly after we ran the executable, indicating that the encoded implant shellcode was successfully transmitted over to the stager.

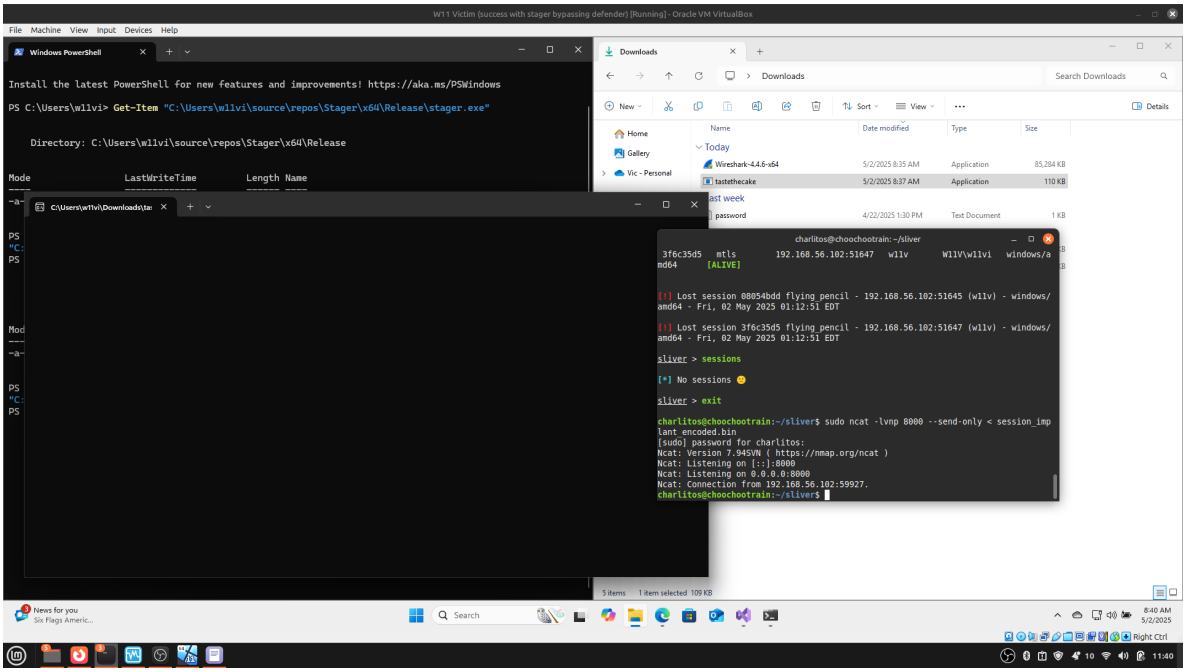

Figure 5: Active netcat listener on the Linux machine that will send the encoded session implant shellcode as soon as a connection is detected.

5 Results and Analysis

Proceeding from the previous figures, we verified that the implant successfully executed on the Windows machine with active Microsoft Defender Antivirus. Figure 6 shows the result of running the stager executable. The `jobs` command lists the active Sliver listeners, and the `sessions` command lists the current active sessions. As shown, an active Sliver session named `flying_pencil` was established, confirming that our implant executed as intended. Additionally, Defender did not react at all—its protections remained fully enabled, yet the session remained alive.

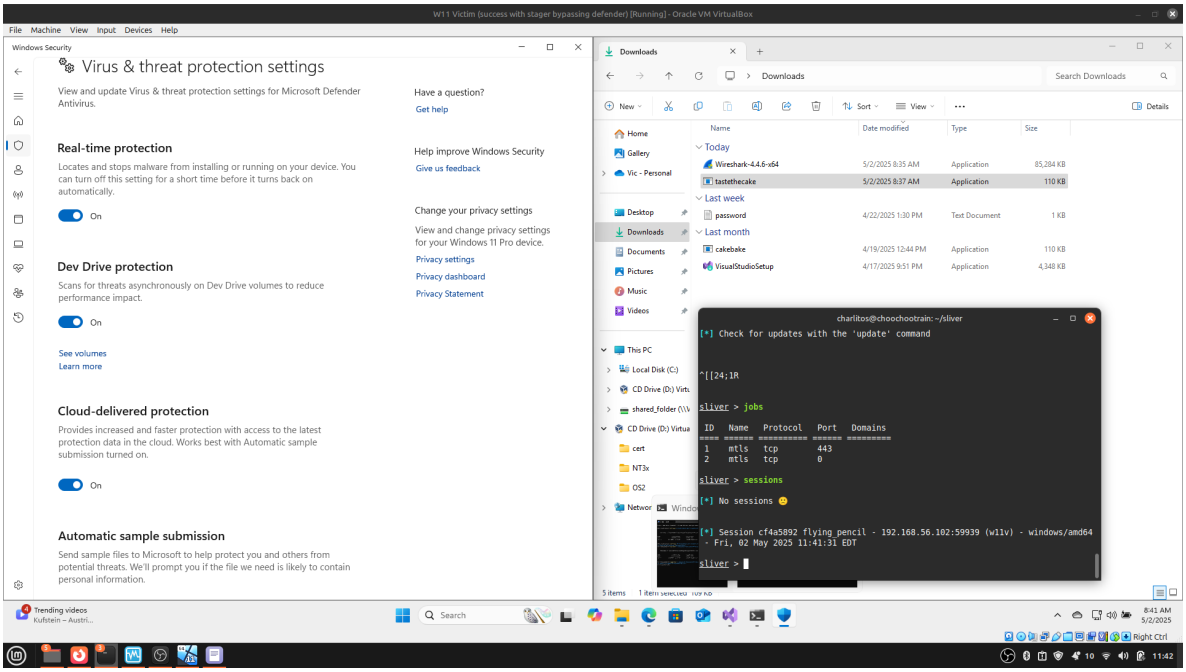

Figure 6: Active session established through the flying_pencil implant, with all Defender protections turned on.

For further confirmation, we issued a few commands through the active session. As shown in Figure 7, executing the `ls` command revealed that we were operating from within the Windows `Downloads` directory, confirming that the implant had successfully integrated with the target environment.

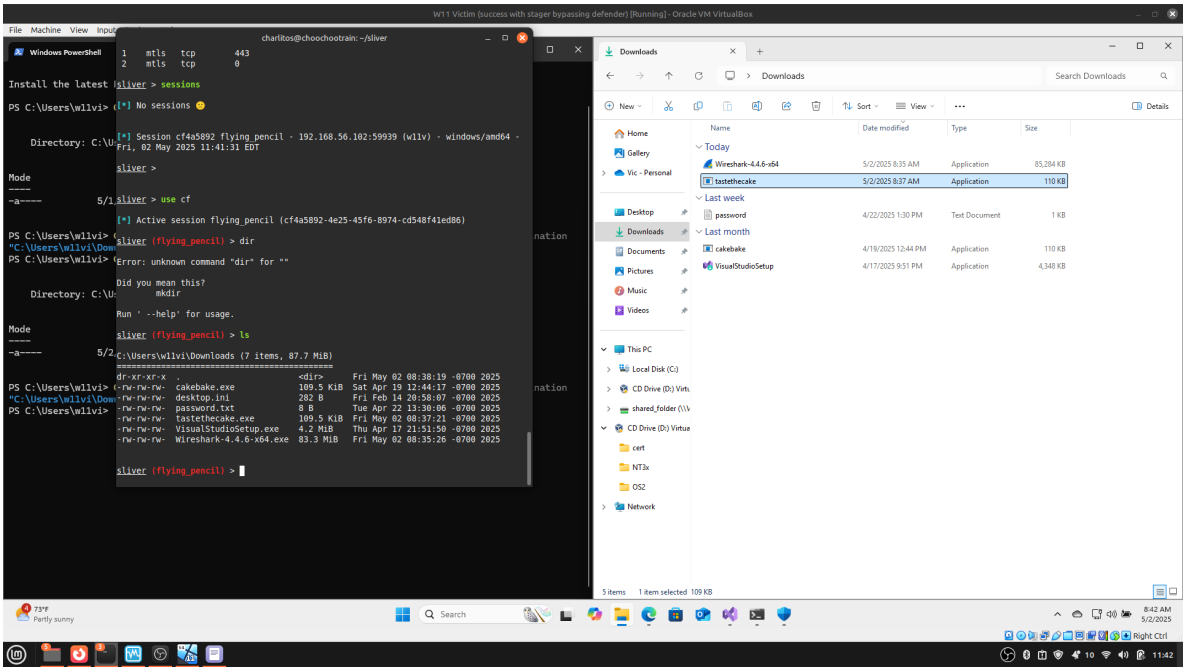

Figure 7: Running `ls` through the Sliver session shows we are indeed in the Windows `Downloads` directory, revealing its contents.

We also ran `whoami` to verify our current user context on the system, shown in Figure 8. The

output confirmed that we were executing commands under a legitimate user account on the Windows system, supplementing the success of the in-memory implant.

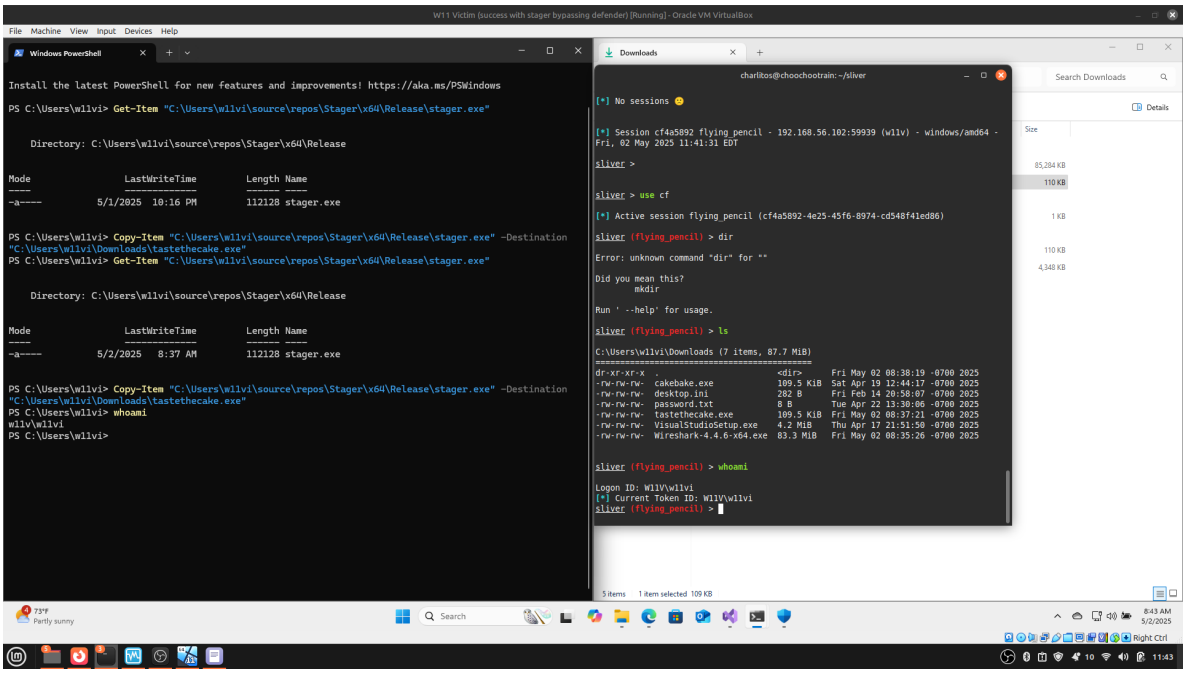

Figure 8: Running `whoami` through the Sliver session reveals the current Windows user.

5.1 Discussion

Although Microsoft Defender actively scans both files on disk and processes in memory, our custom stager was able to execute the implant without detection. This outcome can be attributed to several key evasion factors built into the design and deployment strategy of the stager.

First, the stager was compiled from custom C++ source code and did not match any known static signatures typically used by Defender to detect malware. Signature-based detection relies on identifying patterns, hashes, or structural characteristics known to be associated with malicious software. Although the stager was written to disk in the `Downloads` folder, its footprint was minimal, and it lacked embedded indicators such as known shellcode patterns or suspicious import tables. Therefore, it did not trigger any static detection alerts.

Second, the stager employed self-injection rather than injecting into external processes. By allocating memory and executing the shellcode within its own process context, it avoided behaviors commonly flagged as suspicious, such as cross-process memory manipulation using `WriteProcessMemory` and `CreateRemoteThread`. Defender's behavioral analysis engines tend to prioritize inter-process tampering and known system binaries being targeted for injection. Our approach reduced the likelihood of triggering dynamic detection mechanisms.

Third, runtime behavior was limited in scope and duration. Because the stager executed quickly and communicated with the C2 server without attempting privilege escalation, persistence, or overt reconnaissance, it did not cause any prolonged pattern of suspicious activity. Defender may place less emphasis on low-suspicion processes in its memory scans unless a high-risk indicator is present.

Finally, the payload used by the stager was XOR-encoded and only decoded at runtime. This obfuscation helped lower the chance of Defender identifying the shellcode through entropy or pattern analysis during static inspection. Since the decoded shellcode existed only briefly in memory and within the stager process, memory scanning techniques were unlikely to flag it without specific heuristics targeting such behavior.

Together, these factors contributed to the stager's ability to execute and establish a C2 session without being flagged by Microsoft Defender's real-time protection mechanisms.

Interestingly, when we attempted to use remote process injection—targeting `notepad.exe` or `explorer.exe`—we observed different results. For instance, simply copying the stager binary into the `Downloads` folder triggered a Windows Security pop-up recommending a cloud-based scan, as shown in Figure 9.

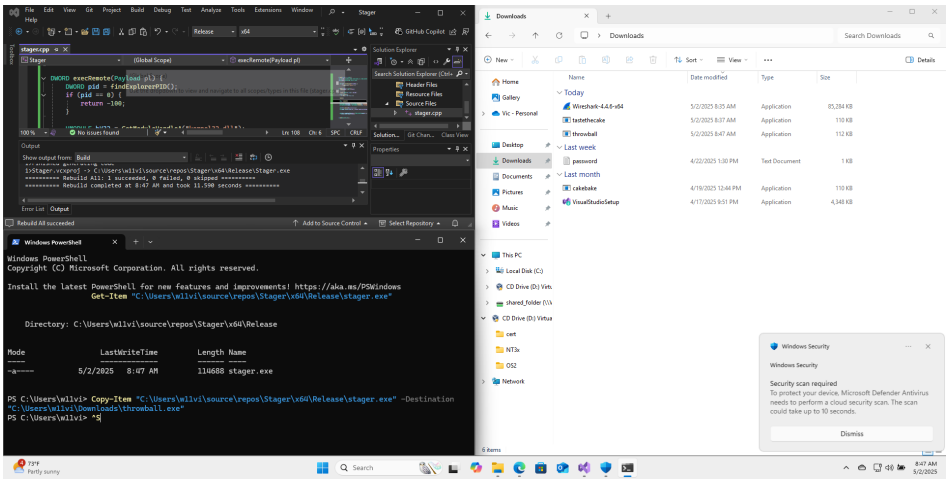

Figure 9: Windows Security prompts a cloud scan after detecting a potential threat when the remote process injection stager is copied to the `Downloads` folder.

Although we could ignore the warning and still run the stager, it was only able to establish a session with the C2 server when we used `explorer.exe` as the target process, and not `notepad.exe`. We believe this is due to Defender's process-specific behavioral detection models. In other words, it was likely anomalous for `notepad.exe`—a lightweight text editor that typically has no network activity—to initiate outbound connections, causing Microsoft Defender to block the behavior or terminate the process silently [40]. By contrast, `explorer.exe` regularly interacts with the network and loads dynamic content, making it a less suspicious target for injection. Nevertheless, this illustrates that remote injection patterns are more likely to be flagged. By contrast, our self-injection variant raised no such alerts and successfully established a session. This suggests that Microsoft Defender has adapted to detect common attack techniques involving remote process manipulation.

While these results suggest that our self-injecting stager is stealthier against detection by Microsoft Defender Antivirus compared to using remote process injection, this does not mean it is fully stealthy in a broader security context. Although it avoided triggering static or behavioral alerts from Defender, the process itself would likely appear suspicious to a human analyst or modern EDR platform. An unfamiliar binary name like `tastethecake.exe`, executing from the `Downloads` folder, performing memory allocation and in-memory shellcode execution, and creating outbound network connections are all strong signs of potentially malicious behavior. Therefore, while the technique proved effective against Defender alone, its stealth is likely to be less effective in more advanced or monitored environments.

6 Ethical Considerations

This project involved the use of offensive security tools, specifically the Sliver C2 framework, to explore post-exploitation evasion techniques against Microsoft Defender AV in a controlled environment. All tests were performed within a virtual machine, with no real-world systems or third-party networks involved.

While both the attacker (Linux) and target (Windows) systems retained internet access via NAT adapters to support tool installation and updates, command and control (C2) traffic was strictly confined to a dedicated `Host-only Adapter` interface within VirtualBox. This configuration ensured that all implant communication occurred over a closed, non-routable subnet (192.168.56.0/24), fully isolated from external networks.

No real-world malware, ransomware, or persistence mechanisms were created. The implants were generated strictly for short-term educational use, focused on command execution and in-memory eva-

sion testing. During the final evaluation phase, Microsoft Defender's automatic sample submission feature was enabled to assess detection by its cloud-based services, with full awareness that submitted payloads could be analyzed by Microsoft.

At no point were live networks, third-party systems, or external targets scanned, probed, or exploited. All activities were strictly confined to a controlled virtual lab environment. This project adheres to established academic ethical standards for cybersecurity research and contributes to the development of practical understanding in post-exploitation defense and detection[41][42].

7 Conclusion

This project demonstrated that a custom-built C++ stager, designed with modern evasion techniques, can successfully bypass Microsoft Defender Antivirus and execute a Sliver implant entirely in memory. By avoiding static imports, using self-injection, dynamically resolving API calls, and obfuscating payloads with XOR encoding, the stager avoided detection at every stage—file scan, memory allocation, and shellcode execution.

7.1 Lessons Learned

Through the development and testing of this custom stager, we observed that Microsoft Defender Antivirus—despite its widespread deployment and continued improvements—remains susceptible to evasion techniques that carefully avoid known malware behaviors. Simple techniques like code obfuscation, XOR encoding, and dynamic API resolution, combined with self-injection and in-memory execution, were enough to bypass detection without the need for kernel-level exploits or complex packers.

We also learned the importance of reducing the footprint of malware artifacts during each stage of execution. The self-injection approach significantly reduced behavioral indicators compared to remote process injection, which raised alarms even before payload execution. This emphasized that stealth is often achieved not by one technique alone, but by a deliberate combination of minimalistic design, runtime obfuscation, and execution tactics that avoid behavioral triggers.

Finally, from a defender's perspective, this project illustrated the limitations of static and signature-based detection, and underscored the importance of enhancing behavioral monitoring, memory inspection, and anomaly detection capabilities. As attackers increasingly adopt fileless techniques, endpoint security solutions must adapt accordingly to maintain effectiveness.

7.2 Future Work

This project can be extended by evaluating evasion techniques against other antivirus solutions and more advanced EDR systems. Another potential avenue of exploration involves studying how long a custom stager can remain undetected in a real-world environment, given that Microsoft Defender Antivirus frequently receives cloud-driven updates and machine learning model retraining. Investigating techniques for sustained evasion, such as using polymorphic stagers that modify their structure or behavior over time, could help achieve long-term stealth and resilience against evolving detection mechanisms. Additionally, future research could examine techniques that target Microsoft Defender Antivirus directly, not merely to evade detection, but to interfere with its ability to analyze or respond to malicious activity. This may include exploring ways to disable or mislead Defender's scanning and behavioral analysis engines, such as tampering with AMSI (Antimalware Scan Interface), manipulating Defender's memory space, or exploiting internal weaknesses to reduce visibility into the stager's execution.

8 Acknowledgements

I would like to thank Caeland Gardner and Erin Freck from the Virginia Tech Information Technology Security Office for their help in the inspiration and brainstorming of this work. I would also like to express my gratitude toward my committee/faculty advisors: Dr. Scott Midkiff, Professor Randy Marchany, and Dr. Haining Wang for their flexibility and support throughout the process. Finally, I

would like to acknowledge Bishop Fox for developing the Sliver C2 framework, and Dominic Breuker for his in-depth blog on Sliver and stagers, which served as the foundation for this research.

References

[1] Adam Goss, "What are c2 frameworks?" December 2025, [Online]. Available: https://www.stationx.net/what-is-a-c2-framework/, Accessed: 2025-04-30.

[2] joe, "Getting started — sliver c2 wiki," May 2023, [Online]. Available: https://github.com/BishopFox/sliver/wiki/Getting-Started/599e6cef2173540928a381864a2322fc89538fdd, Accessed: 2025-04-30.

[3] J. Hunsley, "Developing for multiple platforms with go," April 2014, [Online]. Available: https://techblog.steelseries.com/2014/04/08/multi-platform-development-go.html, Accessed: 2025-04-30.

[4] J. Vijayan, "'sliver' emerges as cobalt strike alternative for malicious c2," August 2022, [Online]. Available: https://www.darkreading.com/vulnerabilities-threats/-sliver-cobalt-strike-alternative-malicious-c2, Accessed: 2025-04-30.

[5] A. Mishra, "Sliver framework customized enhances evasion and bypasses edr detection," April 2025, [Online]. Available: https://gbhackers.com/sliver-framework-customized-enhances-evasion/, Accessed: 2025-04-30.

[6] Bishop Fox, "Sliver: Cross-platform general purpose implant framework written in golang," 2025, [Online]. Available: https://bishopfox.com/tools/sliver, Accessed: 2025-04-30.

[7] BishopFox, "Sliver documentation," 2025, [Online]. Available: https://sliver.sh/docs, Accessed: 2025-04-30.

[8] Z. Done and B. Avdieiev, "Hunting for a sliver in a haystack," April 2024, [Online]. Available: https://www.huntandhackett.com/blog/hunting-for-a-sliver, Accessed: 2025-04-30.

[9] N. S. Rocafort, P. Jennings, and D. S. Team, "Sliver c2: How darktrace provided a sliver of hope," April 2024, [Online]. Available: https://www.darktrace.com/blog/sliver-c2-how-darktrace-provided-a-sliver-of-hope-in-the-face-of-an-emerging-c2-framework, Accessed: 2025-04-30.

[10] L. Castel and M. Antonyan, "Sliver c2 leveraged by many threat actors," April 2024, [Online]. Available: https://www.cybereason.com/blog/sliver-c2-leveraged-by-many-threat-actors, Accessed: 2025-04-30.

[11] MicrosoftDocumentation, "Microsoft defender antivirus on windows," 2024, [Online]. Available: https://learn.microsoft.com/en-us/defender-endpoint/microsoft-defender-antivirus-windows, Accessed: 2025-04-30.

[12] Microsoft, "Advanced technology of microsoft defender antivirus," 2024, [Online]. Available: https://learn.microsoft.com/en-us/defender-endpoint/adv-tech-of-mdav, Accessed: 2025-04-30.

[13] Trend Micro Incorporated, "Yara rules - deep discovery inspector online help," 2021, [Online]. Available: https://docs.trendmicro.com/all/ent/ddi/v5.8/en-us/ddi_5.8_olh/YARA-Rules.html, Accessed: 2025-04-30.

[14] B. LaPorte, "Fileless malware will beat your edr," December 2024, [Online]. Available: https://www.morphisec.com/blog/fileless-malware-attacks/, Accessed: 2025-04-30.

[15] K. Townsend, "Fileless attacks ten times more likely to succeed: Report," November 2017, [Online]. Available: https://www.securityweek.com/fileless-attacks-ten-times-more-likely-succeed-report/, Accessed: 2025-04-30.

[16] Inventive HQ, "Edr vs. antivirus – why traditional security isn't enough," 2024, [Online]. Available: https://inventivehq.com/edr-vs-antivirus-why-traditional-security-isnt-enough/, Accessed: 2025-04-30.

[17] M. ATT&CK, "Process injection (t1055)," April 2025, [Online]. Available: https://attack.mitre.org/techniques/T1055/, Accessed: 2025-04-30.

[18] R3dLevy, "Offensive development with c++: Process injection part ii — practical examples," April 2025, [Online]. Available: https://medium.com/@R3dLevy/offensive-development-with-c-process-injection-part-ii-practical-examples-5bd89fdf00ab, Accessed: 2025-04-30.

[19] Microsoft Developer Network, "Createremotethread function (processthreadsapi.h)," October 2022, [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createremotethread, Accessed: 2025-04-30.

[20] Y. Mikanana, "Power of shellcode: Benefits of shellcode and writing one yourself," November 2023, [Online]. Available: https://medium.com/@yua.mikanana19/power-of-shellcode-benefits-of-shellcode-and-writing-one-yourself-ae052a1d00c8, Accessed: 2025-04-30.

[21] M. ATT&CK, "Obfuscated files or information: Dynamic api resolution (t1027.007)," April 2025, [Online]. Available: https://attack.mitre.org/techniques/T1027/007/, Accessed: 2025-04-30.

[22] Microsoft Developer Network, "Getmodulehandlea function (libloaderapi.h)," February 2023, [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandlea, Accessed: 2025-04-30.

[23] ——, "Getprocaddress function (libloaderapi.h)," February 2024, [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress, Accessed: 2025-04-30.

[24] C. Hummel, "Command and control mechanisms for post-exploitation," M.S. thesis, Louisiana State University and Agricultural and Mechanical College, Baton Rouge, LA, USA, October 2024, [Online]. [Online]. Available: https://repository.lsu.edu/gradschool_theses/6043

[25] L. B. Yehuda and O. Golan, "Bypassing av solution using simple evasion techniques – part 1," September 2023, [Online]. Available: https://medium.com/@lielomer76/bypassing-av-solution-using-simple-evasion-techniques-part-1-160cf90751c0, Accessed: 2025-04-30.

[26] D. Breuker, "Learning sliver c2 (06) – stagers," September 2022, [Online]. Available: https://dominicbreuker.com/post, Accessed: 2025-04-30.

[27] Kaspersky Lab, "How cybercriminals try to bypass antivirus protection," April 2025, [Online]. Available: https://usa.kaspersky.com/resource-center/threats/combating-antivirus, Accessed: 2025-04-30.

[28] C. Billinis, "Bypassing windows defender runtime scanning," May 2020, [Online]. Available: https://labs.withsecure.com/publications/bypassing-windows-defender-runtime-scanning, Accessed: 2025-04-30.

[29] M. ATT&CK, "Obfuscated files or information: Command obfuscation (t1027.010)," April 2025, [Online]. Available: https://attack.mitre.org/techniques/T1027/010/, Accessed: 2025-04-30.

[30] Andrea Lelli. (2018, September) Out of sight but not invisible: Defeating fileless malware with behavior monitoring, amsi, and next-gen av. [Online]. Available: https://www.microsoft.com/en-us/security/blog/2018/09/27/out-of-sight-but-not-invisible-defeating-fileless-malware-with-behavior-monitoring-amsi-and-next-gen-av/,Accessed:2025-04-30.

[31] A. Belak, "The quiet victories and false promises of machine learning in security," October 2022, [Online]. Available: https://cloudsecurityalliance.org/blog/2022/10/24/the-quiet-victories-and-false-promises-of-machine-learning-in-security/, Accessed: 2025-04-30.

[32] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," September 2019, [Online]. Available: https://arxiv.org/abs/1706.06083, Accessed: 2025-04-30.

[33] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *Proceedings of the 2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. Saarbrücken, Germany: IEEE, March 2016, pp. 372–387. [Online]. Available: https://ieeexplore.ieee.org/document/7467366

[34] L. Chen, Y. Ye, and T. Bourlai, "Adversarial machine learning in malware detection: Arms race between evasion attack and defense," in *2017 European Intelligence and Security Informatics Conference (EISIC)*, 2017, pp. 99–106.

[35] U. Verma, Y. Huang, C. Woodward, C. Schmugar, P. P. Ramagopal, and C. Fralick, "Attacking malware detection using adversarial machine learning," in *2022 4th International Conference on Data Intelligence and Security (ICDIS)*, 2022, pp. 40–49.

[36] E. Ginger, "Building a simple custom implant for av bypassing," February 2022, [Online]. Available: https://redheadsec.tech/building-a-simple-custom-implant-for-sliver-shellcode/, Accessed: 2025-04-30.

[37] B. Fox, "Sliver adversary emulation framework," 2025, [Online]. Available: https://github.com/BishopFox/sliver, Accessed: 2025-05-05.

[38] moloch, "Sliver c2 server-side implementation," 2025, [Online]. Available: https://github.com/BishopFox/sliver/blob/master/server/c2/README.md, Accessed: 2025-04-30.

[39] M. Ashraf, "Tryhackme av evasion: Shellcode," November 2024, [Online]. Available: https://medium.com/@Mx0o14/tryhackme-av-evasion-shellcode-585f57e0a87b, Accessed: 2025-04-30.

[40] Microsoft Defender Security Research Team, "Uncovering cross-process injection with windows defender atp," March 2017, [Online]. Available: https://www.microsoft.com/en-us/security/blog/2017/03/08/uncovering-cross-process-injection-with-windows-defender-atp/, Accessed: 2025-05-02.

[41] I. of Electrical and E. Engineers, "Ieee code of ethics," 2020, [Online]. Available: https://www.ieee.org/about/corporate/governance/p7-8.html, Accessed: 2025-04-30.

[42] D. Dittrich and E. Kenneally, "The menlo report: Ethical principles guiding information and communication technology research," U.S. Department of Homeland Security, Tech. Rep., 2012, [Online]. Available: https://www.caida.org/publications/papers/2012/menlo_report_actual_formatted/menlo_report_actual_formatted.pdf, Accessed: 2025-04-30.