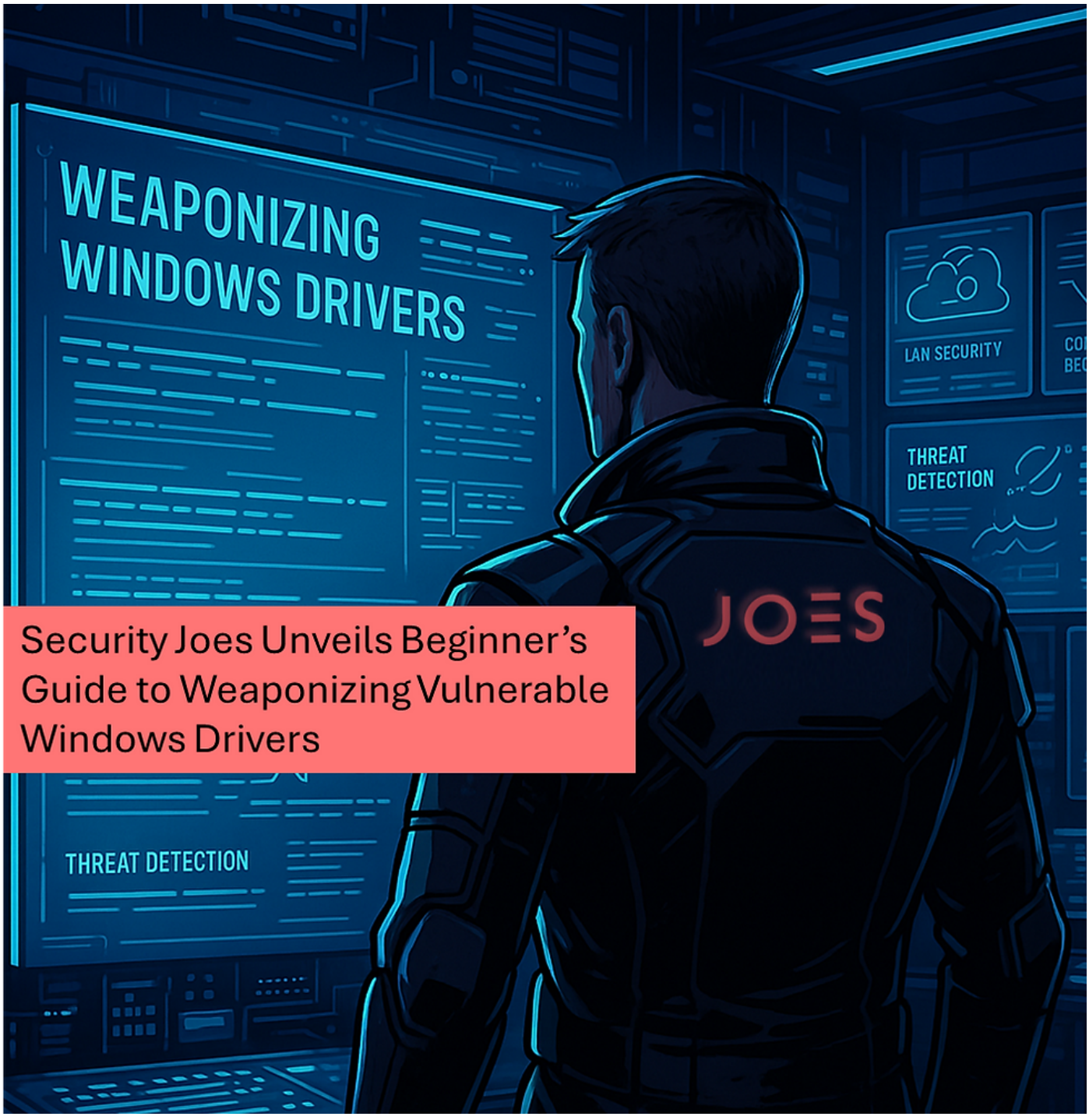
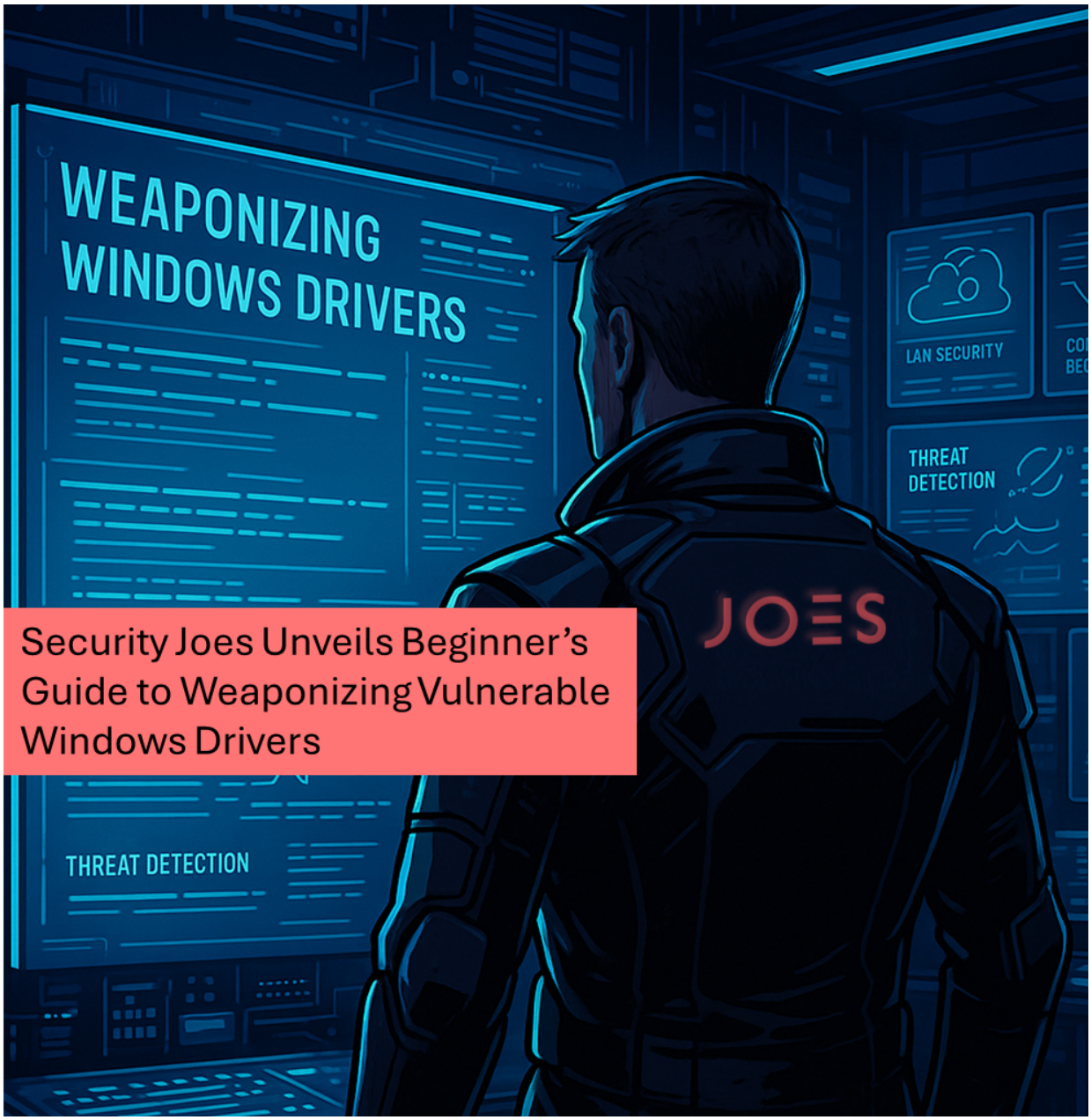
Weaponizing Windows Drivers: A Hacker's Guide for Beginners

Security Joes ⋮ ⋮ 15/07/2025





In the never-ending cat-and-mouse game of cybersecurity, every advancement in defense inevitably drives attackers to evolve their tactics, exploiting new gaps and vulnerabilities. From the early days of signature-based antivirus solutions in the 1980s to today's sophisticated behavioral and machine learning-driven detection tools, the landscape of cyber defense has become increasingly complex and robust. Yet, despite this progress, attackers continue to find creative ways to bypass even the most advanced protections.

One of the most persistent challenges in this arms race lies within the Windows kernel. While userland malware remains common, it is often detected through its interaction with monitored Windows APIs. Kernel-mode threats, on the other hand, operate at a much deeper level, directly interfacing with core components of the operating system, making them significantly stealthier and difficult to detect.

Recognizing this advantage, attackers began developing kernel-mode malware in the early 2000s, prompting Microsoft to implement key countermeasures such as Kernel Patch Protection (PatchGuard), kernel callbacks, and Driver Signature Enforcement (DSE). These mitigations helped curb the spread of rootkits and unauthorized kernel modifications. However, modern attackers have adapted by leveraging a technique known as Bring Your Own Vulnerable Driver (BYOVD), which involves exploiting legitimate but flawed drivers that are already signed and trusted by the system. A 2024 Kaspersky report highlights the growing prevalence of this technique, noting a 23% increase in related attacks.

In this context, understanding how to analyze and exploit vulnerable drivers is critical—not just for offensive security professionals conducting red team operations, but also for defenders aiming to stay ahead of evolving threats.

This article marks the beginning of a four-part playbook designed to guide readers through the process of reverse engineering and exploiting Windows drivers. Starting with static analysis, we will identify vulnerabilities, develop a working exploit, and ultimately demonstrate how such drivers can be weaponized. Whether you're seeking to strengthen defensive strategies or simulate real-world attacks, this series offers a practical foundation for engaging with one of the most complex and high-impact areas of Windows security.

The article in a nutshell:

```
[+] Introduction to the growing threat of kernel-mode attacks and the
limitations of traditional security measures in detecting them.

[+] It highlights the rise of the BYOVD (Bring Your Own Vulnerable Driver)
technique, where attackers exploit signed but flawed drivers to bypass kernel
protections.

[+] Aimed at both red teamers and defenders, the guide begins with static
analysis and sets the foundation for understanding and exploiting kernel-
level vulnerabilities.
```

Security Joes is a multi-layered incident response company strategically located in nine different time-zones worldwide, providing a follow-the-sun methodology to respond to any incident remotely. Security Joes' clients are protected against this threat.

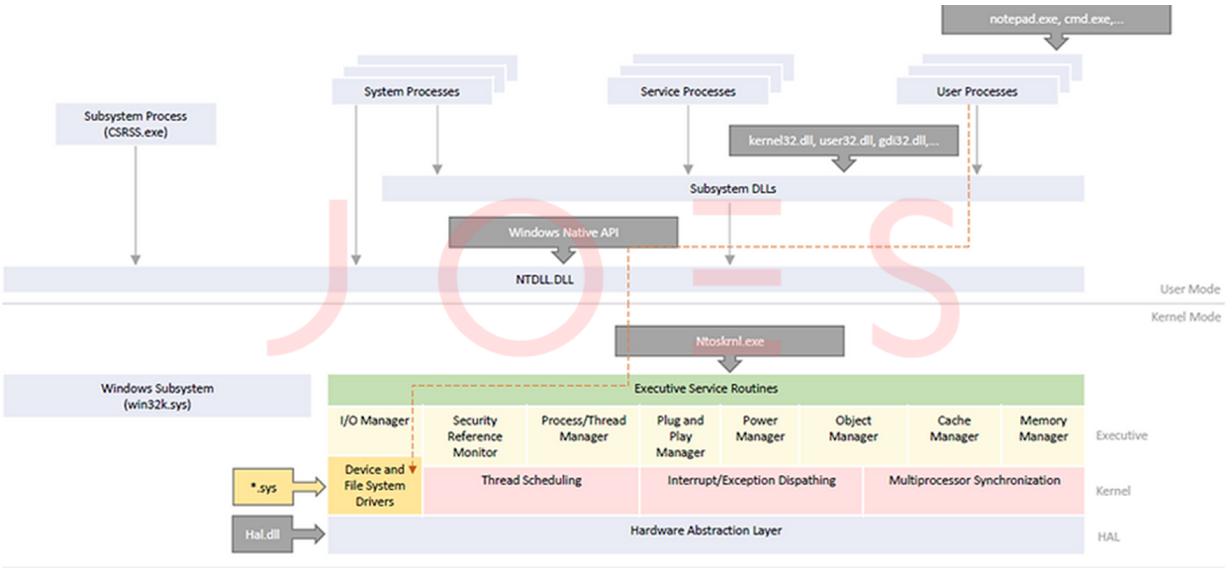
Contact us at response@securityjoes.com for more information about our services and technologies and get additional recommendations to protect yourself against this kind of attack vector.

What is a Windows Driver?

A driver is a software component that can be dynamically loaded or unloaded by the operating system. It interfaces with the Windows kernel and, in many cases, manages hardware resources. However, not all drivers are tied to physical hardware. A useful way to conceptualize a driver is as a container of subroutines that the operating system invokes to perform specific tasks—some hardware-related, others purely software-based.

For example, an Endpoint Detection and Response (EDR) driver may not control any hardware directly but operates by registering callbacks in the kernel to monitor and respond to security-related system events. Because drivers run in kernel mode, they possess high privileges and unrestricted access to system resources. This makes them a high-value target for attackers aiming to escalate privileges, disable security mechanisms such as EDR callbacks, and achieve full control over the system.

The figure below outlines the Windows architecture and its corresponding layers, some of which will be explored in more detail throughout this article.



Windows architecture and its corresponding layers.

Windows Driver Loading Process

For testing and exploitation purposes, it is strongly recommended to install and load drivers under analysis within a virtual machine. This precaution helps avoid compromising or crashing the host system during development or testing.

In Windows, drivers can be installed as services using the *CreateService* API or the built-in service control utility, *sc.exe*. When a driver is registered as a service, a corresponding registry entry is created under:

```
HKLM\SYSTEM\CurrentControlSet\Services
```

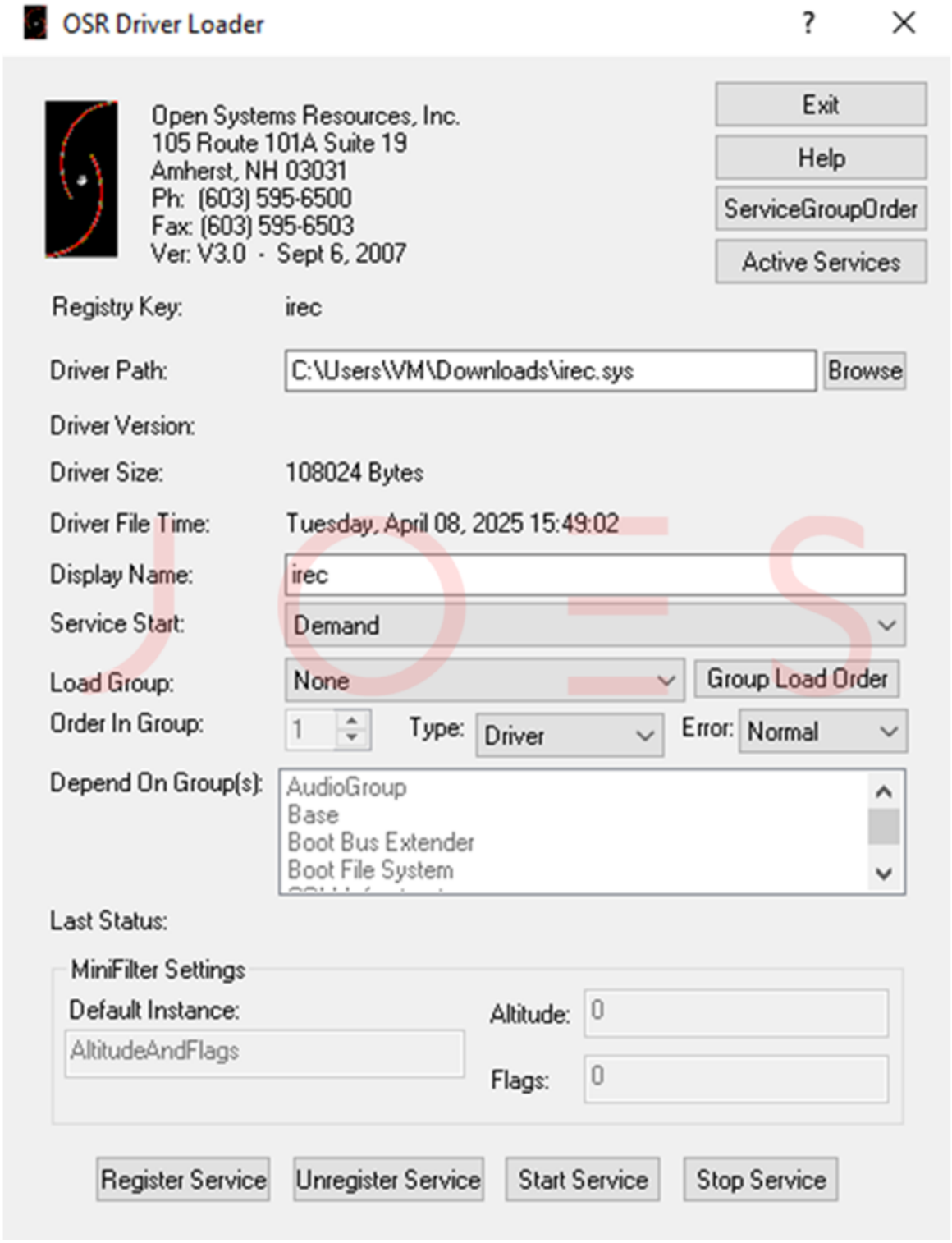
Windows drivers use the *.SYS* file extension and can be installed using *sc.exe* with the following commands:

```
sc create <Service Name> type=kernel binPath=<Driver Path>
sc start <Service Name>
```

While installing a driver using *sc.exe* is straightforward, we'll also demonstrate an alternative method using a more user-friendly tool that may be familiar to some: OSR Driver Loader.

After downloading and extracting the utility, launch the *OSRLOADER.exe* executable. In the **Driver Path** field, specify the full path to the *.SYS* driver file.

Then, click **Register Service**, followed by **Start Service** to load the driver into the system, as shown in the figure below:



Graphical interface of OSR Driver Loader

For testing purposes, we will use the irec.sys driver, which is known to contain documented vulnerabilities that allow a local attacker to execute arbitrary code and escalate privileges, as detailed in CVE-2023-41444. The driver can be downloaded from this link.

Anatomy of Windows Drivers

Before beginning static analysis in IDA Pro, it's essential to develop a basic understanding of the driver's source code written in C. Below is the implementation of the **DriverEntry** function, which acts as the standard entry point for any Windows driver. In kernel mode, **DriverEntry** serves a role similar to the main function in user-mode applications.

When a driver is loaded, its image is mapped into kernel memory, and a DRIVER_OBJECT is created, registered, and partially initialized by the Object Manager. This structure represents the driver within the system and is passed by the I/O Manager to the **DriverEntry** function. Within this function, the driver performs necessary initialization and registers the routines required to handle I/O requests from user-mode applications or other system components.

The **RegistryPath** parameter passed to **DriverEntry** contains the registry path where the driver's configuration settings are stored. However, for the purpose of our analysis, this parameter is not relevant and can be safely ignored.

```
extern "C" NTSTATUS
DriverEntry(
      _In_ PDRIVER_OBJECT DriverObject,
      _In_ PUNICODE_STRING RegistryPath
) {
      ...
      DriverObject->DriverUnload = SecJoesDriverUnload;
      DriverObject->MajorFunction[IRP_MJ_CREATE] = SecJoesCreateClose;
      DriverObject->MajorFunction[IRP_MJ_CLOSE] = SecJoesCreateClose;
      DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
SecJoesIoDeviceControl;
      UNICODE_STRING devName =
RTL_CONSTANT_STRING(L"\\Device\\SecJoesDevice");
      PDEVICE_OBJECT DeviceObject;
      NTSTATUS status = IoCreateDevice(DriverObject, 0, &devName,
      FILE_DEVICE_UNKNOWN, 0, FALSE, &DeviceObject
      );
      if (!NT_SUCCESS(status)) { ... }
      UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\??\\SecJoesDevice");
      status = IoCreateSymbolicLink(&symLink, &devName);
      if (!NT_SUCCESS(status)) {...}
      ...
}
```

The DRIVER_OBJECT structure is a partially opaque data structure, though its full definition is available in the Windows Driver Kit (WDK) header files. While it contains numerous fields, we will focus only on those most relevant to our analysis. Below we describe ach of them with details:

- **DriverUnload**: Holds a pointer to the function responsible for releasing resources allocated by the driver and reversing any initialization performed in **DriverEntry**. If this function is not properly implemented or invoked, it can result in a resource leak that persists until the system is rebooted.

  In the provided code, the cleanup function is **SecJoesDriverUnload**. This callback function adheres to the following format:

```
DRIVER_UNLOAD DriverUnload;

void DriverUnload(
  [in] _DRIVER_OBJECT *DriverObject
)
{...}
```

- **MajorFunction**: This array contains numerous pointers to DRIVER_DISPATCH routines, each responsible for handling a specific type of I/O request—such as Create, Read, Write, and others. These entries are indexed using constants prefixed with IRP_MJ_, which represent the major function codes for different I/O operations.

  In essence, when user-mode applications invoke certain Windows API functions to communicate with the device managed by this driver, those calls are internally translated into I/O Request Packets (IRPs). The I/O Manager then dispatches these IRPs to the appropriate handler function defined in this array. The mapping between IRP codes and their corresponding dispatch routines is illustrated in the table below:

Major Function	Index	Description
IRP_MJ_CREATE	0	Typically invoked for CreateFile or ZwCreateFile calls
IRP_MJ_CLOSE	2	Typically invoked for CloseHandle or ZwClose calls
IRP_MJ_READ	3	Typically invoked for ReadFile or ZwReadFile calls
IRP_MJ_WRITE	4	Typically invoked for WriteFile or ZwWriteFile calls
IRP_MJ_DEVICE_CONTROL	14	Typically invoked for DeviceIoControl or ZwDeviceIoControlFile calls

In the code example above, the **SecJoesCreateClose** function is designated to handle both Create and Close operations, while the **SecJoesIoDeviceControl** function manages **DeviceIoControl** requests. **DeviceIoControl** is a Windows API that allows user-mode applications to send custom I/O control codes (IOCTLs) to kernel-mode drivers. This mechanism enables communication between user-mode programs and drivers, allowing for hardware control or the execution of specialized operations.

Due to its flexibility and direct interaction with driver internals, **DeviceIoControl** is frequently targeted in attacks against Windows drivers. As such, it will be a central focus throughout this playbook.

The corresponding callback function for handling these requests follows the format below:

```
DRIVER_DISPATCH DriverDispatch;

NTSTATUS DriverDispatch(
  [in, out] _DEVICE_OBJECT *DeviceObject,
  [in, out] _IRP *Irp
)
{...}
```

Another key element within the driver's entry point is the **IoCreateDevice** function, which is used to create a device object for the driver. A device object represents either a physical or logical device and defines its characteristics. In Windows, communication is directed not at the driver itself, but at the device it manages—this device becomes the target for all I/O operations initiated from user mode.

The structure representing the device is DEVICE_OBJECT. When a device object is created, the driver can optionally assign a name to it, placing it within the Object Manager's namespace. By default, device objects reside in the *Device* directory, which is not accessible from user space.

In the example above, the device is assigned the name **\Device\SecJoesDevice**, and a DEVICE_OBJECT structure is passed to **IoCreateDevice** for initialization and configuration.

The prototype for the **IoCreateDevice** function is shown below:

```
NTSTATUS IoCreateDevice(
  [in]           PDRIVER_OBJECT  DriverObject,
  [in]           ULONG           DeviceExtensionSize,
  [in, optional] PUNICODE_STRING DeviceName,
  [in]           DEVICE_TYPE     DeviceType,
  [in]           ULONG           DeviceCharacteristics,
  [in]           BOOLEAN         Exclusive,
  [out]          PDEVICE_OBJECT  *DeviceObject
);
```

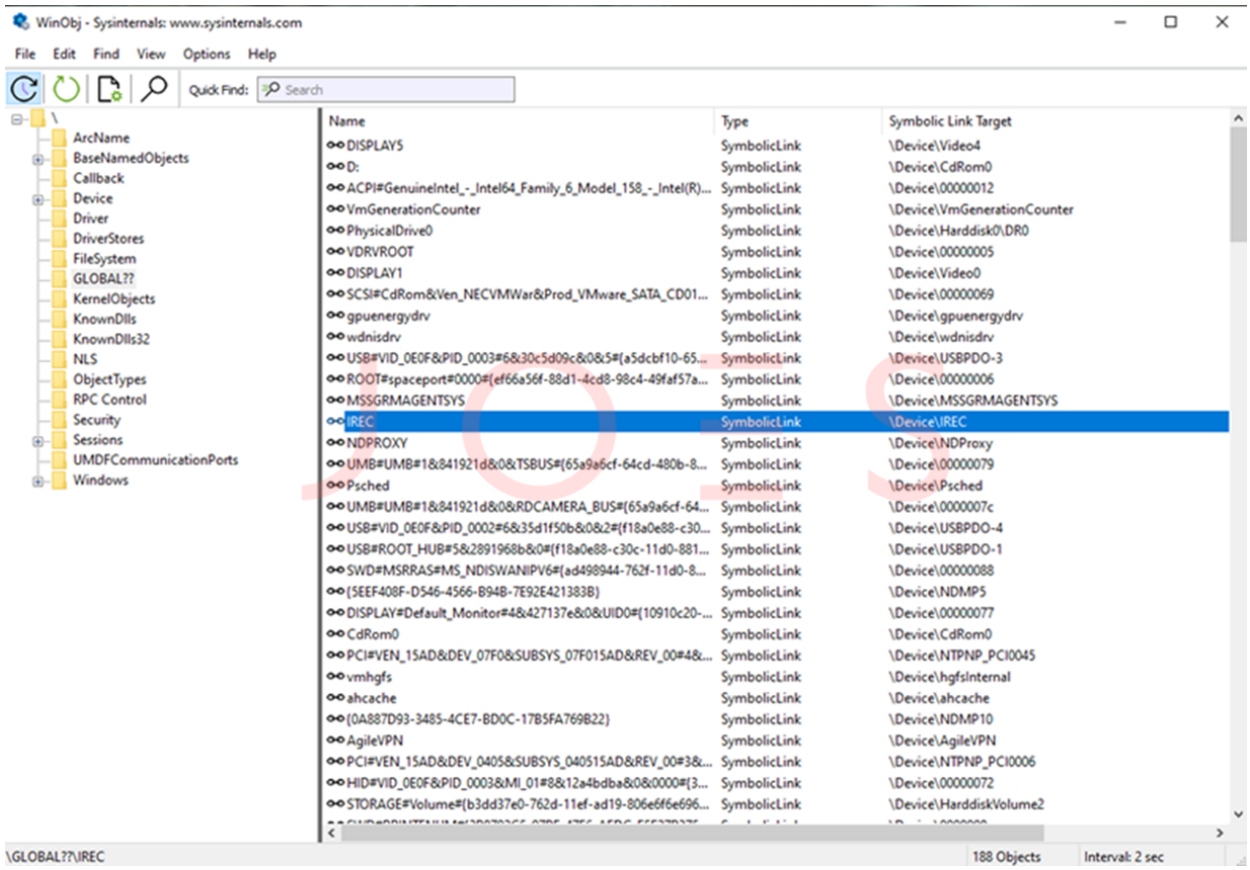
If the driver is intended to allow user-mode applications to access the device it manages, it must create a symbolic link that maps a user-accessible name to the device object located in the \Device directory.

This is typically accomplished using the **IoCreateSymbolicLink** function. In the example code, a symbolic link named \??\SecJoesDevice is created. Although \?? appears to denote a virtual object

directory, in kernel mode it typically resolves to the \GLOBAL?? directory, making the device accessible from user mode under the defined alias.

The concept of a *Directory* in the kernel differs from that of a traditional file system directory. In the kernel context, a Directory is a specialized kernel object that serves as a container for other kernel objects—including additional Directory objects—forming a hierarchical structure managed by the Object Manager. This structure is used to organize, manage, and resolve named kernel objects throughout the system.

Tools such as WinObj and ObjectExplorer allow us to visualize and navigate this hierarchy. For example, using WinObj, you can observe that a symbolic link named IREC appears in the *\GLOBAL??* directory after the *irec.sys* driver is loaded via the OSR Driver Loader. As a practical exercise, you can explore the *\Device* directory to verify the presence of the IREC device object.



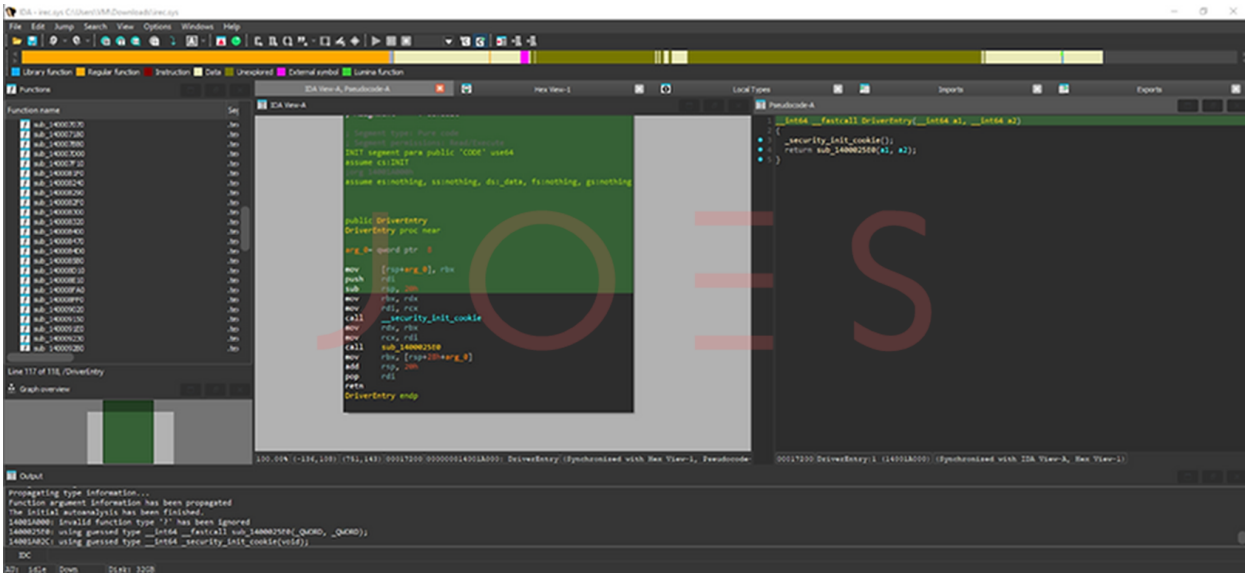
WinObj GUI where the Windows drivers hierarchy is presented.

Later in this playbook, we will confirm that these objects were indeed created by the driver during static analysis using IDA Pro.

Static Analysis

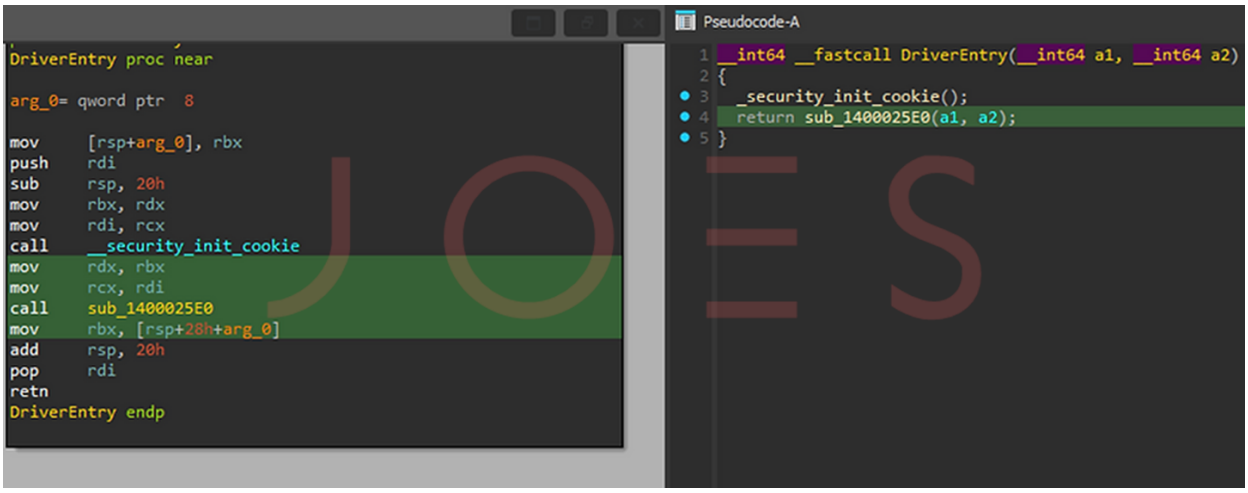
Static analysis plays a crucial role in identifying potential vulnerabilities in software, especially in low-level components like drivers. Through static code inspection, which involves reviewing the program's structure and behavior without running it, analysts can identify unsafe functions, insecure coding patterns, and logic flaws that could be exploited by attackers. Disassemblers are essential tools in this process, as they allow researchers to inspect compiled binaries, understand program behavior, and trace how data flows through the code—even in the absence of source code.

You’re free to use any disassembler of your choice; however, be aware that the available features, interface, and command sets may differ significantly between tools. For this guide, we’ll be using the freeware version of IDA Pro, one of the most widely used disassemblers for reverse engineering. Once the driver is loaded into IDA Pro, you’ll be presented with the following screen:



Initial screen loaded in IDA Pro, after loading the driver.

Once IDA completes its initial analysis of the driver file, we can identify the DriverEntry function. Throughout this article, we will primarily focus on the Pseudocode-A tab, which presents the decompiled code, rather than the IDA View-A tab — unless we need to examine specific Assembly instructions directly. The figure below shows both views side by side. We can observe that, initially, the parameter types for the DriverEntry function are unknown. To resolve this, we will use the Type Libraries provided by IDA Pro.



Assembly instructions (left) and pseudocode (right) in driver entry point as seen in IDA Pro.

Type Libraries are collections of high-level type information specific to certain platforms and compilers, used by IDA Pro and its decompiler to enhance code analysis. They include definitions such as function prototypes, typedefs, structures, enums, and constants, allowing IDA to accurately identify data types and improve the readability and precision of the decompiled output.

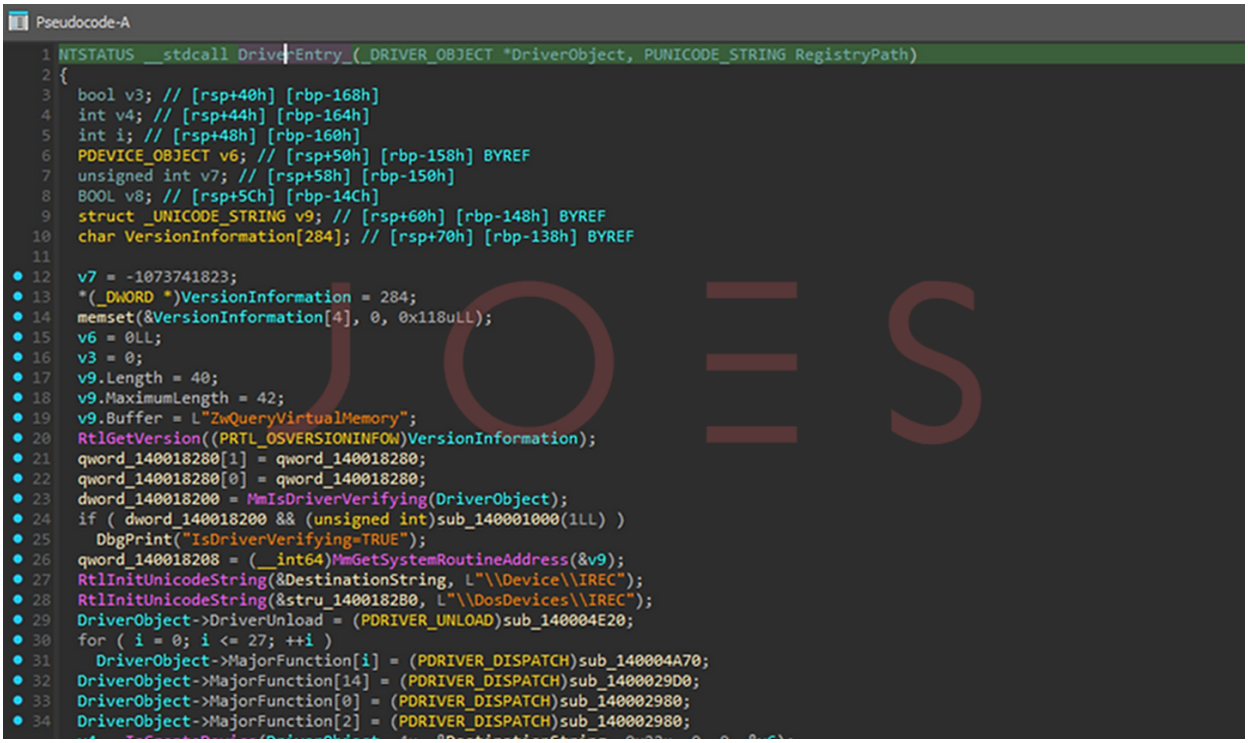
We're going to apply the types used by the Windows kernel. To do this, go to *View* > *Open subviews* > *Type Libraries (or press Shift+F11)*. In the Type Libraries window, right-click on an empty area and select Load type library... (or press Insert). Then, choose the **ntddk64** library and click OK. The parameter types for **DriverEntry** will be applied automatically, as shown below.

Driver pseudocode after loading the type libraries in IDA Pro.

Before applying the type definitions, we noticed something unusual: the function **sub_1400025E0**, which previously took two parameters, is now showing only one. We'll fix this by explicitly setting the correct function prototype and examining its behavior. Since execution flow is handed directly to this function from **DriverEntry**, We'll rename it to **DriverEntry_**.

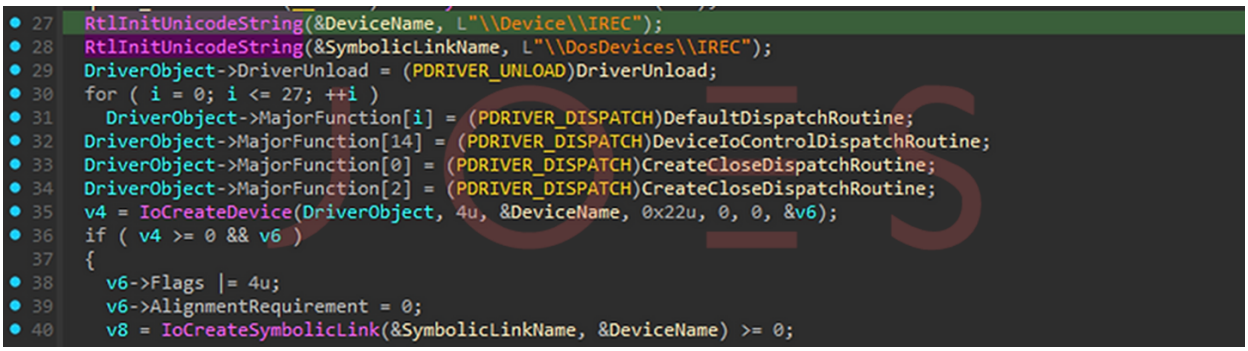
The function's code is shown below. We can immediately identify the use of several callback functions assigned to the MajorFunction array, as well as the initialization of two strings on lines 27 and 28. As we'll see later, these strings are used by the **IoCreateDevice** and **IoCreateSymbolicLink** functions. We can also observe, on line 29, which function is responsible for releasing the resources allocated by **DriverEntry**. It's worth noting one small detail: the symbolic link's name uses the DosDevices prefix

instead of ??. In this context, the DosDevices directory can be considered equivalent to ??. For more information, refer to this link.



Pseudocode of function **DriverEntry_** as seen in IDA Pro.

Based on the information we've gathered and with some understanding of the code, we can rename certain functions and variables within the function to make the analysis clearer and easier to follow. The result is shown below:



Driver's pseudocode after renaming functions to simplify analysis.

The code above, implemented in the **DriverEntry** function, is responsible for creating a device named “IREC” using the **IoCreateDevice** function and establishing a symbolic link with the same name via **IoCreateSymbolicLink**. During the driver's initialization, it also registers the unload routine and sets up the functions that handle system requests, which are configured in the MajorFunction array. Each entry in this array holds the address of a specific routine designed to handle a particular operation requested by user-mode applications. Further details about these routines will be discussed in the next article.

As previously mentioned, the function responsible for handling **DeviceIoControl** requests is a common source of vulnerabilities in drivers and will be the main focus of our next article.

Conclusion

In this article, we covered some essential concepts for analyzing vulnerabilities in our driver and began our static analysis using IDA Pro. We also loaded type definitions into IDA and performed an initial examination of the **DriverEntry** function. In the upcoming chapters, we'll focus on the routine that handles **DeviceIoControl** requests, dive into new concepts, identify a vulnerability, develop an exploit for it, and ultimately weaponize the irec.sys driver.

The world of Windows driver development is vast and complex, and this series only scratches the surface. If you're curious to explore more about drivers, we highly recommend the books *Windows Internals and Windows Kernel Programming*—both are indispensable references for deeper learning.