Lord Of The Ring0 - Part 1 | Introduction


idov31.github.io/2022/07/14/lord-of-the-ring0-p1.html

July 14, 2022

Introduction

This blog post series isn't a thing I normally do, this will be more like a journey that I document during the development of my project Nidhogg. In this series of blogs (which I don't know how long will it be), I'll write about difficulties I encountered while I'm developing Nidhogg and tips & tricks for everyone that wants to start creating a stable kernel mode driver in 2022.

This series will be about WDM type of kernel drivers, developing in VS2019. To install it, you can follow the guide in MSDN. I highly recommend for you test EVERYTHING in a virtual machine to avoid crashing your computer.

Without further delays - Let's start!

Kernel Drivers In 2022

The first question you might ask yourself is: How can kernel driver help me in 2022? There are a lot of 1337 things that I can do for the user mode without the pain of developing and consistently crashing my computer.

From a red team perspective, I think that there are several things that a kernel driver can give that user mode can't.

- Being an efficient backdoor with extremely evasive persistency.
- Do highly privileged operations without the dependency of LPE exploit or privileged user.
- Easily evade AV / EDR hooks.
- Be able to hide your implant without suspicious user-mode hooks.

From a blue team perspective, you can log more events and block suspicious operations with methods you won't be able to do in the user mode.

- Create a driver to monitor and log specific events (like Sysmon) specially crafted to meet your organization's needs.
- Create kernel mode hooks to find advanced rootkits and malware.
- Provide kernel mode protection to your blue agents (such as OSQuery, Wazuh, etc.).

NOTE: This blog series will focus more on the red team part but I'll also add the blue team perspective as one affects the other.

Basic driver structure

Like any good programmer, we will start with creating a basic driver to print famous words with an explanation of the basics.

```
#include <ntddk.h>

extern "C"
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);
    DriverObject->DriverUnload = MyUnload;
    KdPrint(("Hello World!\n"));
    return STATUS_SUCCESS;
}

void MyUnload(PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);
    KdPrint(("Goodbye World!\n"));
}
```

This simple driver will print “Hello World!” and “Goodbye World!” when it’s loaded and unloaded. Since the parameter RegistryPath is not used, we can use UNREFERENCED_PARAMETER to optimize the variable.

Every driver needs to implement at least two of the functions mentioned above.

The DriverEntry is the first function that is called when the driver is loaded and it is very much like the main function for user-mode programs, except it gets two parameters:

- DriverObject: A pointer to the driver object.
- RegistryPath: A pointer to a UNICODE_STRING structure that contains the path to the driver’s registry key.

The DriverObject is an important object that will serve us a lot in the future, its definition is:

```
typedef struct _DRIVER_OBJECT {
  CSHORT             Type;
  CSHORT             Size;
  PDEVICE_OBJECT     DeviceObject;
  ULONG              Flags;
  PVOID              DriverStart;
  ULONG              DriverSize;
  PVOID              DriverSection;
  PDRIVER_EXTENSION  DriverExtension;
  UNICODE_STRING     DriverName;
  PUNICODE_STRING    HardwareDatabase;
  PFAST_IO_DISPATCH  FastIoDispatch;
  PDRIVER_INITIALIZE DriverInit;
  PDRIVER_STARTIO    DriverStartIo;
  PDRIVER_UNLOAD     DriverUnload;
  PDRIVER_DISPATCH   MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
} DRIVER_OBJECT, *PDRIVER_OBJECT;
```

But I'd like to focus on the MajorFunction: This is an array of important functions that the driver can implement for IO management in different ways (direct or with IOCTLs), handling IRPs and more.

We will use it for the next part of the series but for now, keep it in mind. (A little tip: Whenever you encounter a driver and you want to know what it is doing make sure to check out the functions inside the MajorFunction array).

To finish the most basic initialization you will need to do one more thing - define the DriverUnload function. This function will be responsible to stop callbacks and free any memory that was allocated.

When you finish your driver initialization you need to return an NT_STATUS code, this code will be used to determine if the driver will be loaded or not.

Testing a driver

If you tried to copy & paste and run the code above, you might have noticed that it's not working.

By default, windows do not allow to load of self-signed drivers, and surely not unsigned drivers, this was created to make sure that a user won't load a malicious driver and by that give an attacker even more persistence and privileges on the attacked machine.

Luckily, there is a way to bypass this restriction for testing purposes, to do this run the following command from an elevated cmd: After that restart, your computer and you should be able to load the driver.

To test it out, you can use Dbgview to see the output (don't forget to compile to debug to see the KdPrint's output). To load the driver, you can use the following command:

```
sc create DriverName type= kernel binPath= C:\Path\To\Driver.sys
sc start DriverName
```

And to unload it:

```
sc stop DriverName
```

You might ask yourself now, so how an attacker can deploy a driver? This can be done in several ways:

- The attacker has found/generated a certificate (expiration date doesn't matter).
- The attacker has allowed test signing (just like we did now).
- The attacker has a vulnerable driver with 1day that allows loading drivers.
- The attacker has a zero-day that allows load drivers.

Just not so long ago when Nvidia was breached a signature was leaked and used by threat actors.

Resources

When we will continue to dive into the series, I will use a lot of references from the following amazing resources:

- Windows Kernel Programming.
- Windows Internals Part 7.
- MSDN (I know I said amazing, I lied here).

And you can check out the following repositories for drivers examples:

Conclusion

This blog post is maybe short but is the start of the coming series of blog posts about kernel drivers and rootkits specifically. Another one, more detailed, will come out soon!

I hope that you enjoyed the blog and I'm available on Twitter, Telegram and by Mail to hear what you think about it! This blog series is following my learning curve of kernel mode development and if you like this blog post you can check out Nidhogg on GitHub.