Emulation of Kernel Mode Rootkits With Speakeasy

fireeye.com/blog/threat-research/2021/01/emulation-of-kernel-mode-rootkits-with-speakeasy.html

Threat Research

Andrew Davis

Jan 20, 2021

10 mins read

Threat Research

In August 2020, we released a blog post about how the Speakeasy emulation framework can be used to emulate user mode malware such as shellcode. If you haven't had a chance, give the post a read today.

In addition to user mode emulation, Speakeasy also supports emulation of kernel mode Windows binaries. When malware authors employ kernel mode malware, it will often be in the form of a device driver whose end goal is total compromise of an infected system. The malware most often doesn't interact with hardware and instead leverages kernel mode to fully compromise the system and remain hidden.

Challenges With Dynamically Analyzing Kernel Malware

Ideally, a kernel mode sample can be reversed statically using tools such as disassemblers. However, binary packers just as easily obfuscate kernel malware as they do user mode samples. Additionally, static analysis is often expensive and time consuming. If our goal is to automatically analyze many variants of the same malware family, it makes sense to dynamically analyze malicious driver samples.

Dynamic analysis of kernel mode malware can be more involved than with user mode samples. In order to debug kernel malware, a proper environment needs to be created. This usually involves setting up two separate virtual machines as debugger and debugee. The malware can then be loaded as an on-demand kernel service where the driver can be debugged remotely with a tool such as WinDbg.

Several sandbox style applications exist that use hooking or other monitoring techniques but typically target user mode applications. Having similar sandbox monitoring work for kernel mode code would require deep system level hooks that would likely produce significant noise.

Driver Emulation

Emulation has proven to be an effective analysis technique for malicious drivers. No custom setup is required, and drivers can be emulated at scale. In addition, maximum code coverage is easier to achieve than in a sandbox environment. Often, rootkits may expose malicious functionality via I/O request packet (IRP) handlers (or other callbacks). On a normal Windows system these routines are executed when other applications or devices send input/output requests to the driver. This includes common tasks such as reading, writing, or sending device I/O control (IOCTLs) to a driver to execute some type of functionality.

Using emulation, these entry points can be called directly with doped IRP packets in order to identify as much functionality as possible in the rootkit. As we discussed in the first Speakeasy blog post, additional entry points are emulated as they are discovered. A driver's DriverMain entry point is responsible for initializing a function dispatch table that is called to handle I/O requests. Speakeasy will attempt to emulate each of these functions after the main entry point has completed by supplying a dummy IRP. Additionally, any system threads or work items that are created are sequentially emulated in order to get as much code coverage as possible.

Emulating a Kernel Mode Implant

In this blog post, we will show an example of Speakeasy's effectiveness at emulating a real kernel mode implant family publicly named Winnti. This sample was chosen despite its age because it transparently implements some classic rootkit functionality. The goal of this post is not to discuss the analysis of the malware itself as it is fairly antiquated. Rather, we will focus on the events that are captured during emulation.

The Winnti sample we will be analyzing has SHA256 hash c465238c9da9c5ea5994fe9faf1b5835767210132db0ce9a79cb1195851a36fb and the original file name tcprelay.sys. For most of this post, we will be examining the emulation report generated by Speakeasy. Note: many techniques employed by this 32-bit rootkit will not work on modern 64-bit versions of Windows due to Kernel Patch Protection (PatchGuard) which protects against modification of critical kernel data structures.

To start, we will instruct Speakeasy to emulate the kernel driver using the command line shown in Figure 1. We instruct Speakeasy to create a full memory dump (using the "-d" flag) so we can acquire memory later. We supply the memory tracing flag ("-m") which will log all memory reads and writes performed by the malware. This is useful for detecting things like hooking and direct kernel object manipulation (DKOM).

Command line used to emulate the malicious driver

Figure 1: Command line used to emulate the malicious driver

Speakeasy will then begin emulating the malware’s DriverEntry function. The entry point of a driver is responsible for setting up passive callback routines that will service user mode I/O requests as well as callbacks used for device addition, removal, and unloading. Reviewing the emulation report for the malware’s DriverEntry function (identified in the JSON report with an “ep_type” of “entry_point”), shows that the malware finds the base address of the Windows kernel. The malware does this by using the ZwQuerySystemInformation API to locate the base address for all kernel modules and then looking for one named “ntoskrnl.exe”. The malware then manually finds the address of the PsCreateSystemThread API. This is then used to spin up a system thread to perform its actual functionality. Figure 2 shows the APIs called from the malware's entry point.

Figure 2: Key functionality in the tcprelay.sys entry point

Hiding the Driver Object

The malware attempts to hide itself before executing its main system thread. The malware first looks up the “DriverSection” field in its own DRIVER_OBJECT structure. This field holds a linked list containing all loaded kernel modules and the malware attempts to unlink itself to hide from APIs that list loaded drivers. In the “mem_access” field in the Speakeasy report shown in Figure 3, we can see two memory writes to the DriverSection entries before and after itself which will remove itself from the linked list.

Memory write events representing the tcprelay.sys malware attempting to unlink itself in order to hide

Figure 3: Memory write events representing the tcprelay.sys malware attempting to unlink itself in order to hide

As noted in the original Speakeasy blog post, when threads or other dynamic entry points are created at runtime, the framework will follow them for emulation. In this case, the malware created a system thread and Speakeasy automatically emulated it.

Moving on to the newly created thread (identified by an “ep_type” of “system_thread”), we can see the malware begin its real functionality. The malware begins by enumerating all running processes on the host, looking for the service controller process named services.exe. It's important to note that the process listing that gets returned to the emulated samples is configurable via JSON config files supplied at runtime. For more information on these configuration options please see the Speakeasy README on our GitHub repository. An example of this configurable process listing is shown in Figure 4.

Process listing configuration field supplied to Speakeasy

Figure 4: Process listing configuration field supplied to Speakeasy

Pivoting to User Mode

Once the malware locates the services.exe process, it will attach to its process context and begin inspecting user mode memory in order to locate the addresses of exported user mode functions. The malware does this so it can later inject an encoded, memory-resident DLL into the services.exe process. Figure 5 shows the APIs used by the rootkit to resolve its user mode exports.

Logged APIs used by tcprelay.sys rootkit to resolve exports for its user mode implant

Figure 5: Logged APIs used by tcprelay.sys rootkit to resolve exports for its user mode implant

Once the exported functions are resolved, the rootkit is ready to inject the user mode DLL component. Next, the malware manually copies the in-memory DLL into the services.exe process address space. These memory write events are captured and shown in Figure 6.

Memory write events captured while copying the user mode implant into services.exe

Figure 6: Memory write events captured while copying the user mode implant into services.exe

A common technique that rootkits use to execute user mode code involves a Windows feature known as Asynchronous Procedure Calls (APC). APCs are functions that execute asynchronously within the context of a supplied thread. Using APCs allows kernel mode applications to queue code to run within a thread’s user mode context. Malware often wants to inject into user mode since much of the common functionality (such as network communication) within Windows can be more easily accessed. In addition, by running in user mode, there is less risk of being detected in the event of faulty code bug-checking the entire machine.

In order to queue an APC to fire in user mode, the malware must locate a thread in an “alertable” state. Threads are said to be alertable when they relinquish their execution quantum to the kernel thread scheduler and notify the kernel that they are able to dispatch

APCs. The malware searches for threads within the services.exe process and once it detects one that’s alertable it will allocate memory for the DLL to inject then queue an APC to execute it.

Speakeasy emulates all kernel structures involved in this process, specifically the executive thread object (ETHREAD) structures that are allocated for every thread on a Windows system. Malware may attempt to grovel through this opaque structure to identify when a thread’s alertable flag is set (and therefore a valid candidate for an APC). Figure 7 shows the memory read event that was logged when the Winnti malware manually parsed an ETHREAD structure in the services.exe process to confirm it was alertable. At the time of this writing, all threads within the emulator present themselves as alertable by default.

Event logged when the tcprelay.sys malware confirmed a thread was alertable

Figure 7: Event logged when the tcprelay.sys malware confirmed a thread was alertable

Next, the malware can execute any user mode code it wants using this thread object. The undocumented functions KeInitializeApc and KeInsertQueueApc will initialize and execute a user mode APC respectively. Figure 8 shows the API set that the malware uses to inject a user mode module into the services.exe process. The malware executes a shellcode stub as the target of the APC that will then execute a loader for the injected DLL. All of this can be recovered from the memory dump package and analyzed later.

Logged APIs used by tcprelay.sys rootkit to inject into user mode via an APC

Figure 8: Logged APIs used by tcprelay.sys rootkit to inject into user mode via an APC

Network Hooks

After injecting into user mode, the kernel component will attempt to install network obfuscation hooks (presumably to hide the user mode implant). Speakeasy tracks and tags all memory within the emulation space. In the context of kernel mode emulation, this includes all kernel objects (e.g. Driver and Device objects, and the kernel modules themselves).

Immediately after we observe the malware inject its user mode implant, we see it begin to attempt to hook kernel components. This was confirmed during static analysis to be used for network hiding.

The memory access section of the emulation report reveals that the malware modified the netio.sys driver, specifically code within the exported function named NsiEnumerateObjectsAllParametersEx. This function is ultimately called when a user on the system runs the “netstat” command and it is likely that the malware is hooking this function in order to hide connected network ports on the infected system. This inline hook was identified by the event captured in Figure 9.

Inline function hook set by the malware to hide network connections

Figure 9: Inline function hook set by the malware to hide network connections
In addition, the malware hooks the Tcpip driver object in order to accomplish additional network hiding. Specifically, the malware hooks the IRP_MJ_DEVICE_CONTROL handler for the Tcpip driver. User mode code may send IOCTL codes to this function when querying for

active connections. This type of hook can be easily identified with Speakeasy by looking for memory writes to critical kernel objects as shown in Figure 10.

Memory write event used to hook the Tcpip network driver

Figure 10: Memory write event used to hook the Tcpip network driver

System Service Dispatch Table Hooks

Finally, the rootkit will attempt to hide itself using the nearly ancient technique of system service dispatch table (SSDT) patching. Speakeasy allocates a fake SSDT so malware can interact with it. The SSDT is a function table that exposes kernel functionality to user mode code. The event in Figure 11 shows that the SSDT structure was modified at runtime.

SSDT hook detected by Speakeasy

Figure 11: SSDT hook detected by Speakeasy

If we look at the malware in IDA Pro, we can confirm that the malware patches the SSDT entry for the ZwQueryDirectoryFile and ZwEnumerateKey APIs that it uses to hide itself from file system and registry analysis. The SSDT patch function is shown in Figure 12.

File hiding SSDT patching function shown in IDA Pro

Figure 12: File hiding SSDT patching function shown in IDA Pro

After setting up these hooks, the system thread will exit. The other entry points (such as the IRP handlers and DriverUnload routines) in the driver are less interesting and contain mostly boilerplate driver code.

Acquiring the Injected User Mode Implant

Now that we have a good idea what the driver does to hide itself on the system, we can use the memory dumps created by Speakeasy to acquire the injected DLL discussed earlier. Opening the zip file we created at emulation time, we can find the memory tag referenced in Figure 6. We quickly confirm the memory block has a valid PE header and it successfully loads into IDA Pro as shown in Figure 13.

Injected user mode DLL recovered from Speakeasy memory dump

Figure 13: Injected user mode DLL recovered from Speakeasy memory dump

Conclusion

In this blog post, we discussed how Speakeasy can be effective at automatically identifying rootkit activity from the kernel mode binary. Speakeasy can be used to quickly triage kernel binaries that may otherwise be difficult to dynamically analyze. For more information and to check out the code, head over to our GitHub repository.