Red Team Tactics: Combining Direct System Calls and sRDI to bypass AV/EDR | Outflank

By Cornelis

Published: 2019-06-19 · Archived: 2026-04-05 17:57:58 UTC

In this blog post we will explore the use of *direct system calls*, restore hooked API calls and ultimately combine this with a shellcode injection technique called *sRDI*. We will combine these techniques in proof of concept code which can be used to create a LSASS memory dump using Cobalt Strike, while not touching disk and evading AV/EDR monitored user-mode API calls.

As companies grow in their cybersecurity maturity level, attackers also evolve in their attacking capabilities. As a red team we try to offer the best value to our customers, so we also need to adapt to more advanced tactics and techniques attackers are using to bypass modern defenses and detection mechanisms. Recent malware research shows that there is an increase in malware that is using direct system calls to evade user-mode API hooks used by security products. So time for us to sharpen our offensive tool development skills.

Source code of the PoC can be found here:

https://github.com/outflanknl/Dumpert

What are direct system calls?

In order to understand what system calls really are, we first have to dig a little bit into Operating System architecture, specifically Windows.

If you are old (not like me… 😉) and have a MS-DOS background, you probably remember that a simple application crash could result in a complete system crash. This was due to the fact that the operating system was running in *real mode*, which means that the processor is running in a mode in which no memory isolation and protection is applied. A bad program or bug could result in a complete crash of the Operating System due to critical system memory corruption, as there was no restriction in what memory regions could be accessed or not.

This all changed with newer processors and Operating Systems supporting the so-called *protected mode*. This mode introduced many safeguards and could protect the system from crashes by isolating running programs from each other using virtual memory and privilege levels or rings. On a Windows system two of these rings are actually used. Application are running in *user-mode*, which is the equivalent of ring 3 and critical system components like the kernel and device drivers are running in *kernel-mode* which corresponds to ring 0.

Using these protection rings makes sure that applications are isolated and cannot directly access critical memory sections and system resources running in *kernel-mode*. When an application needs to execute a privileged system operation, the processor first needs to switch into ring 0 to handover the execution flow into *kernel-mode*. This is where *system calls* come into place.

Let’s demonstrate this privilege mode switching while monitoring a notepad.exe process and saving a simple text file:

WriteFile call stack in Process Monitor .

The screenshot shows the program flow (call stack) from the notepad.exe process when we save a file. We can see the Win32 API WriteFile call following the Native API NtWriteFile call (more on APIs later).

For a program to save a file on disk, the Operating System needs access to the filesystem and device drivers. These are privileged operations and not something the application itself should be allowed to do. Accessing device drivers directly from an application could result in very bad things. So, the last API call before entering *kernel-mode* is responsible for pulling the dip switch into kernel land.

The CPU instruction for entering *kernel-mode* is the *syscall* instruction (at least on x64 architecture, which we will discuss in this blog only). We can see this in the following WinDBG screenshot, which shows the unassembled NtWriteFile instruction:

Disassembled NtWriteFile API call in WinDBG.

The NtWriteFile API from ntdll.dll is responsible for setting up the relevant function call arguments on the stack, then moving the system call number from the NtWriteFile call in the EAX register and executing the *syscall* instruction. After that, the CPU will jump into *kernel-mode* (ring 0). The kernel uses the dispatch table (SSDT) to find the right API call belonging to the system call number, copies the arguments from the *user-mode* stack into the *kernel-mode* stack and executes the kernel version of the API call (in this case ZwWriteFile). When the kernel routines are finished, the program flow will return back into *user-mode* almost the same way, but will return the return values from the kernel API calls (for example a pointer to received data, or a handle to a file).

This (user-mode) is also the place where many security products like AV, EDR and sandbox software put their hooks, so they can detour the execution flow into their engines to monitor and intercept API calls and block anything suspicious. As you have seen in the disassembled view of the NtWriteFile instruction you may have noticed that it only uses a few assembly instructions, from which the *syscall number* and the *syscall* instruction itself are the most important. The only important thing before executing a direct system call is that the stack is setup correctly with the expected arguments and using the right calling convention.

So, having this knowledge… why not execute the system calls directly and bypass the Windows and Native API, so that we also bypass any user-mode hooks that might be in place? Well this is exactly what we are going to do, but first a little bit more about the Windows programming interfaces.

The Windows programming interfaces

In the following screenshot we see a high-level overview of the Windows OS Architecture:

For a *user-mode* application to interface with the underlying operating system, it uses an application programming interface (API). If you are a Windows developer writing C/C++ application, you would normally use the Win32 API. This is Microsoft's documented programming interfaces which consists of several DLLs (so called Win32 subsystem DLLs).

Underneath the Win32 API sits the Native API (ntdll.dll), which is actually the real interface between the *user-mode* applications and the underlying operating system. This is the most important programming interface but "not officially" documented and should be avoided by programmers in most circumstances.

The reason why Microsoft has put another layer on top of the Native API is that the real magic occurs within this Native API layer as it is the lowest layer between user-mode and the kernel. Microsoft probably decided to shield off the documented APIs using an extra layer, so they could make architectural OS changes without affecting the Win32 programming interface.

So now we know a bit more about *system calls* and the Windows programming APIs, let's see how we can actually skip the programming APIs and invoke the APIs directly using their system call number or restore potentially hooked API calls.

Using system calls directly

We already showed how to disassemble native API calls to identify the corresponding *system call* numbers. Using a debugger this could take a lot of time. So, the same can be done using IDA (or Ghidra) by opening a copy of ntdll.dll and lookup the needed function:

Disassembled NtWriteFile API call in IDA.

One slight problem… system call numbers change between OS versions and sometimes even between service pack/built numbers.

Fortunately @j00ru from Google project Zero comes to the rescue with his online system call tables.

j00ru did an amazing job keeping up with all system call numbers in use by different Windows versions and between builds. So now we have a great resource to look up all the system calls we want to use.

In our code, we want to invoke the system calls directly using assembly. Within Visual Studio we can enable assembly code support using the *masm* build dependency, which allows us to add .asm files and code within our project.

Assembly system call functions in .asm file.

All we need to do is gather OS version information from the system we are using and create references between the native API function definitions and OS version specific system call functions in assembly language. For this we can use the Native API RtlGetVersion routine and save this information into a version info structure.

Reference function pointers based on OS info.

Exported OS specific assembly functions + native API function definitions.

Now we can use the system call functions in our code as if they are normal native API functions:

Using ZwOpenProcess systemcall function as a Native API call.

Restoring hooked API calls with direct system calls

Writing advanced malware that only uses direct system calls and completely evades user-mode API calls is practically impossible or at least extremely cumbersome. Sometimes you just want to use an API call in your malicious code. But what if somewhere in the call stack there is a user-mode hook by AV/EDR? Let’s have a look how we can remove the hook using direct system calls.

Basic user-mode API hooks by AV/EDR are often created by modifying the first 5 bytes of the API call with a jump (JMP) instruction to another memory address pointing to the security software. The technique of unhooking this method has already been documented within two great blog posts by @SpecialHoang, and by MDsec's Adam Chester and Dominic Chell.

If you study these methods carefully, you will notice the use of API calls such as VirtualProtectEx and WriteProcessMemory to unhook Native API functions. But what if the first API calls are hooked and monitored already somewhere in the call stack? Inception, get it? Direct system calls to the rescue!

In the PoC code we created we basically use the same unhooking technique by restoring the first 5 bytes with the original assembly instructions, including the system call number. The only difference is that the API calls we use to unhook the APIs are direct systems call functions (ZwProtectVirtualMemory and ZwWriteVirtualMemory).

Using direct system call function to unhook APIs.

Proof of concept

In our operations we sometimes need Mimikatz to get access to credentials, hashes and Kerberos tickets on a target system. Endpoint detection software and threat hunting instrumentation are pretty good in detection and prevention of Mimikatz nowadays. So, if you are in an assessment and your scenario requires to stay under the radar as much as possible, using Mimikatz on an endpoint is not best practice (even in-memory). Also, dumping LSASS memory with tools such as procdump is often caught by modern AV/EDR using API hooks.

So, we need an alternative to get access to LSASS memory and one option is to create a memory dump of the LSASS process after unhooking relevant API functions. This technique was also documented in @SpecialHoang blog.

As a proof of concept, we created a LSASS memory dump tool called "*Dumpert*". This tool combines direct system calls and API unhooking and allows you to create a LSASS minidump. This might help bypassing defenses of modern AV and EDR products.

The minidump file can be used in Mimikatz to extract credential information without running Mimikatz on the target system.

Mimikatz minidump import.

Of course, dropping executable files on a target is probably something you want to avoid during an engagement, so let’s take this a step further…

sRDI – Shellcode Reflective DLL Injection

If we do not want to touch disk, we need some sort of injection technique. We can create a reflective loadable DLL from our code, but reflective DLL injection leaves memory artefacts behind that can be detected. My colleague @StanHacked recently pointed me to an interesting DLL injection technique called shellcode Reflective DLL Injection.

sRDI allows for the conversion of DLL files to position independent shellcode. This technique is developed by Nick Landers (@monoxgas) from Silent Break Security and is basically a new version of RDI.

Some advantages of using sRDI instead of standard RDI:

- You can convert any DLL to position independent shellcode and use standard shellcode injection techniques.
- Your DLL does not need to be reflection-aware as the reflective loader is implemented in shellcode outside of your DLL.
- Uses proper Permissions, no massive RWX blob.
- Optional PE Header Cleaning.

More detailed information about sRDI can be found in this blog.

Let our powers combine!

Okay with all the elements in place, let's see if we can combine these elements and techniques and create something more powerful that could be useful during Red Team operations:

- We created a DLL version of the "dumpert" tool using the same direct system calls and unhooking techniques. This DLL can be run standalone using the following command line: "***rundll32.exe C:\Dumpert\Outflank-Dumpert.dll,Dump***", but in this case we are going to convert it to a sRDI shellcode.

- Compile the DLL version using Visual Studio and turn it into a position independent shellcode. This can be done using the *ConvertToShellcode.py* script from the sRDI project: ***"python3 ConvertToShellcode.py Outflank-Dumpert.dll"***
- To inject the shellcode into a remote target, we can use Cobalt Strike's *shinject* command. Cobalt Strike has a powerful scripting language called aggressor script which allows you to automate this step. To make this easier we provided an aggressor script which enables a "dumpert" command in the beacon menu to do the dirty job.

- The dumpert script uses shinject to inject the sRDI shellcode version of the dumpert DLL into the current process (to avoid CreateRemoteThread API). Then it waits a few seconds for the lsass minidump to finish and finally download the minidump file from the victim host.
- Now you can use Mimikatz on another host to get access to the lsass memory dump including credentials, hashes e.g. from our target host. For this you can use the following command: *"**sekurlsa::minidump C:\Dumpert\dumpert.dmp**"*

Conclusion

Malware that evades security product hooks is increasing and we need to be able to embed such techniques in our projects.

In this blog we used references between the native API function definitions and OS version specific system call functions in assembly. This allows us to use direct system calls function as if they were normal Native API functions. We combined this technique together with an API unhooking technique to create a minidump from the LSASS process and used sRDI in combination with Cobalt Strike to inject the *dumpert* shellcode into memory of a target system.

Detecting malicious use of system calls is difficult. Because user-mode programming interfaces are bypassed, the only place to look for malicious activity is the kernel itself. But with kernel PatchGuard protection it is infeasible for security products to create hooks or modification in the running kernel.

I hope this blogpost is useful in understanding the advanced techniques attackers are using nowadays and provides a useful demonstration on how to emulate these techniques during Red Team operations.If you have any feedback or additional ideas, let me know.

Lastly, in order to help other red teams easily implement these techniques and more, we've developed Outflank Security Tooling (OST), a broad set of evasive tools that allow users to safely and easily perform complex tasks. If

you’re interested in seeing the diverse offerings in OST, we recommend scheduling an expert led demo.
