Threat Intelligence

Staying Hidden on the Endpoint: Evading Detection with Shellcode

October 10, 2019

Mandiant

Written by: Evan Pena, Casey Erikson

True red team assessments require a secondary objective of avoiding detection. Part of the glory of a successful red team assessment is not getting detected by anything or anyone on the system. As modern Endpoint Detection and Response (EDR) products have matured over the years, the red teams must follow suit. This blog post will provide some insights into how the FireEye Mandiant Red Team crafts payloads to bypass modern EDR products and get full command and control (C2) on their victims' systems.

Shellcode injection or its execution is our favorite method for launching our C2 payload on a victim system; but what is shellcode? Michael Sikorski defines shellcode as a "…term commonly used to describe any piece of self-contained executable code" (Practical Malware Analysis). Most commercial Penetration Testing Frameworks such as Empire, Cobalt Strike, or Metasploit have a shellcode generator built into the tool. The shellcode generator is generally in either a binary format or hex format depending on whether you generate it as raw output or as an application source.

Why do we use shellcode for all our payloads?

The use of shellcode in our red team assessment payloads allows us to be incredibly flexible in the type of payload we use. Shellcode runners can be in written in a wide range of programming languages that can be incorporated into many types of payloads. This flexibility allows us to customize our payloads to support the specific needs of our clients and of any given situation that may arise during a red team assessment. Since shellcode can be launched from inside a payload or injected into depending on the scenario and technology in place in the target environment. Several techniques exist for obfuscating shellcode, such as encryption and custom encoding, that make it difficult for EDR products to detect shellcode from commercial C2 tools on its own. The flexibility and evasive properties of shellcode are the primary reason that we rely heavily on shellcode based payloads during red team assessments.

Shellcode Injection Vs. Execution

One of the most crucial parts of any red team assessment is developing a payload that will successfully, reliably, and stealthily run on the target system. Payloads can either execute shellcode from within its own process or inject shellcode into the address space of another process that will ultimately execute the shellcode. For the purposes of this blog post we'll refer to shellcode injection as shellcode executed inside a remote process and shellcode execution as shellcode executed inside the payload process.

Shellcode injection is one technique that red teams and malicious attackers use to avoid detection from EDR products and network defenders. Additionally, many EDR products implement detections based on expected behavior of windows processes. For example, an attacker that executes Mimikatz from the context of an arbitrary process, let's say DefinitelyNotEvil.exe, may get detected or blocked outright because the EDR tool does not expect that process to access lsass.exe. However, by injecting into a windows process, such as svchost.exe, that regularly touches lsass.exe, it may be possible to bypass these detections because the EDR product sees this as an expected behavior.

In this blog post, we'll cover three different techniques for running shellcode.

- CreateThread
- CreateRemoteThread
- QueueUserAPC

Each of these techniques corresponds to a Windows API function that is responsible for the allocation of a thread to the shellcode, ultimately resulting in the shellcode being run. CreateThread is a technique used for shellcode execution while CreateRemoteThread and QueueUserAPC are forms of shellcode injection.

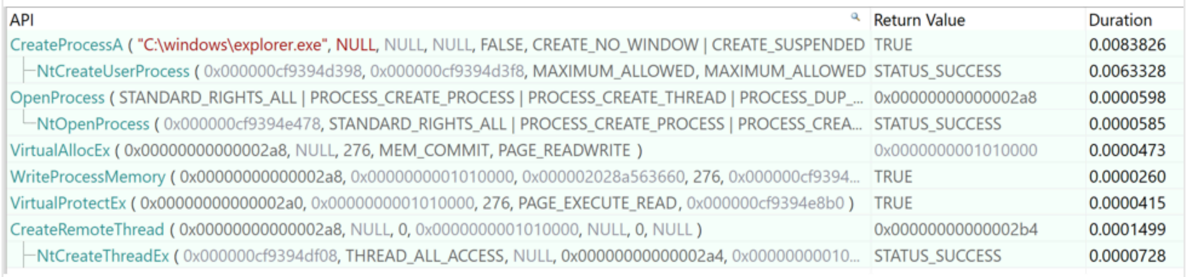
The following is a high-level outline of the process for running shellcode with each of the three different techniques.

CreateThread

3. Modify the protections of the newly allocated memory to allow execution of code from within that memory space
4. Create a thread with the base address of the allocated memory segment
5. Wait on the thread handle to return

CreateRemoteThread

1. Get the process ID of the process to inject into
2. Open the target process
3. Allocate executable memory within the target process
4. Write shellcode into the allocated memory
5. Create a thread in the remote process with the start address of the allocated memory segment

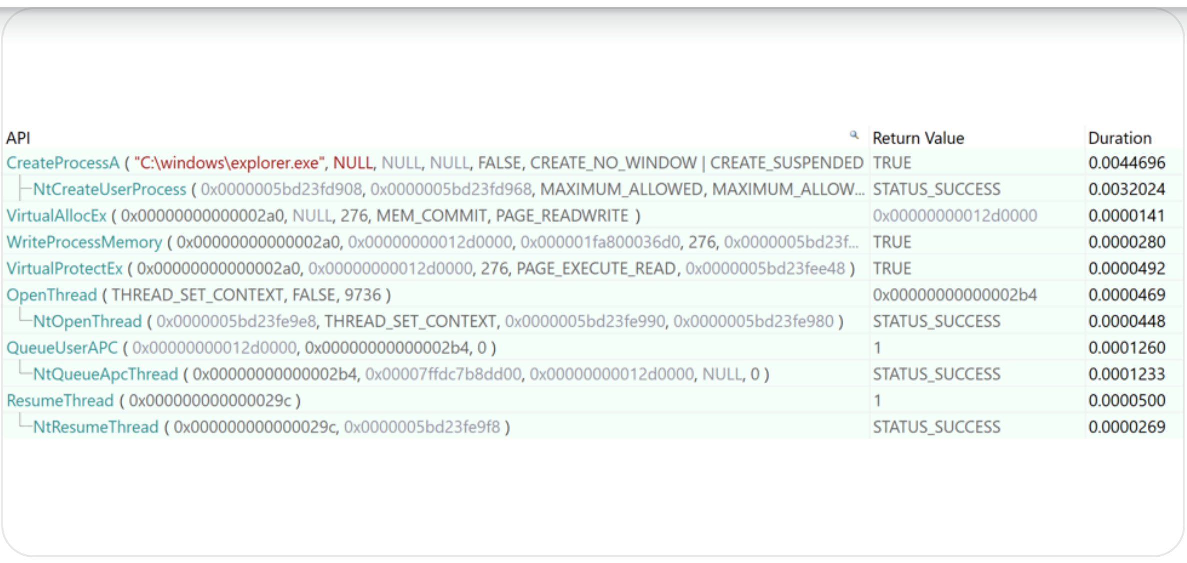


API	Return Value	Duration
CreateProcessA ("C:\windows\explorer.exe", NULL, NULL, NULL, FALSE, CREATE_NO_WINDOW | CREATE_SUSPENDED	TRUE	0.0083826
├NtCreateUserProcess (0x000000cf9394d398, 0x000000cf9394d3f8, MAXIMUM_ALLOWED, MAXIMUM_ALLOWED	STATUS_SUCCESS	0.0063328
OpenProcess (STANDARD_RIGHTS_ALL | PROCESS_CREATE_PROCESS | PROCESS_CREATE_THREAD | PROCESS_DUP_...	0x00000000000002a8	0.0000598
└NtOpenProcess (0x000000cf9394e478, STANDARD_RIGHTS_ALL | PROCESS_CREATE_PROCESS | PROCESS_CREA...	STATUS_SUCCESS	0.0000585
VirtualAllocEx (0x00000000000002a8, NULL, 276, MEM_COMMIT, PAGE_READWRITE)	0x0000000001010000	0.0000473
WriteProcessMemory (0x00000000000002a8, 0x0000000001010000, 0x000002028a563660, 276, 0x000000cf9394...	TRUE	0.0000260
VirtualProtectEx (0x00000000000002a0, 0x0000000001010000, 276, PAGE_EXECUTE_READ, 0x000000cf9394e8b0)	TRUE	0.0000415
CreateRemoteThread (0x00000000000002a8, NULL, 0, 0x0000000001010000, NULL, 0, NULL)	0x00000000000002b4	0.0001499
├NtCreateThreadEx (0x000000cf9394df08, THREAD_ALL_ACCESS, NULL, 0x00000000000002a4, 0x00000000010...	STATUS_SUCCESS	0.0000728

Figure 1: Windows API calls for CreateRemoteThread injection

QueueUserAPC

1. Get the process ID of the process to inject into
2. Open the target process
3. Allocate memory within the target process
4. Write shellcode into the allocated memory
5. Modify the protections of the newly allocated memory to allow execution of code from within that memory space
6. Open a thread in the remote process with the start address of the allocated memory segment
7. Submit thread to queue for execution when it enters an “alertable” state
8. Resume thread to enter “alertable” state



API	Return Value	Duration
CreateProcessA ("C:\windows\explorer.exe", NULL, NULL, NULL, FALSE, CREATE_NO_WINDOW | CREATE_SUSPENDED	TRUE	0.0044696
├NtCreateUserProcess (0x0000005bd23fd908, 0x0000005bd23fd968, MAXIMUM_ALLOWED, MAXIMUM_ALLOW...	STATUS_SUCCESS	0.0032024
VirtualAllocEx (0x00000000000002a0, NULL, 276, MEM_COMMIT, PAGE_READWRITE)	0x000000000012d0000	0.0000141
WriteProcessMemory (0x00000000000002a0, 0x000000000012d0000, 0x000001fa800036d0, 276, 0x0000005bd23f...	TRUE	0.0000280
VirtualProtectEx (0x00000000000002a0, 0x000000000012d0000, 276, PAGE_EXECUTE_READ, 0x0000005bd23fee48)	TRUE	0.0000492
OpenThread (THREAD_SET_CONTEXT, FALSE, 9736)	0x00000000000002b4	0.0000469
└NtOpenThread (0x0000005bd23fe9e8, THREAD_SET_CONTEXT, 0x0000005bd23fe990, 0x0000005bd23fe980)	STATUS_SUCCESS	0.0000448
QueueUserAPC (0x000000000012d0000, 0x00000000000002b4, 0)	1	0.0001260
└NtQueueApcThread (0x00000000000002b4, 0x00007ffdc7b8dd00, 0x000000000012d0000, NULL, 0)	STATUS_SUCCESS	0.0001233
ResumeThread (0x000000000000029c)	1	0.0000500
└NtResumeThread (0x000000000000029c, 0x0000005bd23fe9f8)	STATUS_SUCCESS	0.0000269

Figure 2: Windows API calls for QueueUserAPC injection

Command Execution

Let's break down what we've talked about so far:

- Malicious code is your shellcode – the stage 0 or stage 1 code that is truly going to do the malicious work.
- Standard "shellcode runner" application which executes your code via either injection or execution. Most everyone writes their own shellcode runner, so we don't necessarily deem this as true malware, the real malware is the shellcode itself.

Now that we've covered all that, we need a method to execute the code you compiled. Generally, this is either an executable (EXE) or a Dynamic Link Library (DLL). The Red Team prefers using Living Off the Land Binaries (lolbins) **commands** which will execute our compiled **code**.

The reason we can take advantage of lolbins is because of unmanaged exports. At a high level, when an executable calls a DLL it is looking for a specific export within the DLL to execute the code within that export. If the export is not properly protected, then you can craft your own DLL with the export name you know the executable is looking for and run your arbitrary code; which in this case will be your shellcode runner.

Putting It All Together

We set out to develop a shellcode runner DLL that takes advantage of lolbins through unmanaged exports while also providing the flexibility to execute both injected and non-injected shellcode without a need to update the code base. This effort resulted in a C# shellcode runner called DueDLLigence, for which the source code can be found at the GitHub page.

The DueDLLigence project provides a quick and easy way to switch between different shellcode techniques described previously in this blog post by simply switching out the value of the global variable shown in Figure 3.

The DueDLLigence DLL contains three unmanaged exports inside of it. These exports can be used with the Rasautou, Control, and Coregen native Windows commands as described in Figure 4. Note: The shellcode that is in the example will only pop calc.

Native Windows Executable	**Required Export Name**	**Syntax Used To Run**
Rasautou	Powershell	rasautou –d {full path to dll} –p powershell –a a –e e
Control	Cplapplet	Rename compiled "dll" extension to "cpl" and just double click it!
MSIExec	Dllunregisterserver	msiexec /z {full path to dll}

Figure 4: DueDLLigence execution outline

When you open the source code you will find the example uses the exports shown in Figure 5.

Figure 5: Source code for exported entry points

The first thing you should do is generate your own shellcode. An example of this is shown in Figure 6, where we use Cobalt Strike to generate raw shellcode for the "rev_dns" listener. Once that is complete, we run the base64 -w0 payload.bin > [outputFileName] command in Linux to generate the base64 encoded version of the shellcode as shown in Figure 7.

Figure 6: Shellcode generation

Figure 7: Converting shellcode to base64

Then you simply replace the base64 encoded shellcode on line 58 with the base64'd version of your own x86 or x64 shellcode. The screenshot in Figure 6 generated an x86 payload, you will need to check the "use x64 payload" box to generate an x64 payload.

using a different project it doesn't work properly. You can reinstall opening the NuGet package manager console shown in Figure 8 and running the Install-Package UnmanagedExports -Version 1.2.7 command.

Figure 8: Open NuGet Package Manager

After you have reinstalled the Unmanaged exports library and replaced the base64 encoded shellcode on line 58 then you are ready to compile! Go ahead and build the source and look for your DLL in the bin folder. We strongly suggest that you test your DLL to ensure it has the proper exports associate with it. Visual Studio Pro comes with the Dumpbin.exe utility which you can run against your DLL to view the exports as shown in Figure 9.

Figure 9: Dumpbin.exe output

You can expand the list as much as you want with more lolbin techniques found over at the GitHub page.

We prefer to remove the unmanaged exports that are not going to be used with the respective payload that was generated so there is a smaller footprint in the payload. In general, this is good tradecraft when crafting payloads or writing code. In our industry we have the principle of least privilege, well this is the principle of least code!

Modern Detections for Shellcode Injection

Despite all the evasive advantages that shellcode offers, there is hope when it comes to detecting shellcode injection. We looked at several different methods for process injection.

In our shellcode runner, the shellcode injection techniques (CreateRemoteThread and QueueUserAPC) spawn a process in a suspended state and then inject shellcode into the running process. Let's say we choose the process to inject into as explorer.exe and our payload will run with MSIExec. This will create a process tree where cmd.exe will spawn msiexec.exe which will in turn spawn explorer.exe.

Figure 10: Process tree analysis

In an enterprise environment it is possible to collect telemetry data with a SIEM to determine how often, across all endpoints, the cmd.exe -> msiexec.exe -> explorer.exe process tree occurs. Using parent-child

API hooking is commonly used by EDR and AV products to monitor and for detect the use of Windows API calls that are commonly used by malware authors. Utilizing kernel routines such as PsSetCreateProcessNotifyRoutine(Ex) and PsSetCreateThreadNotifyRoutine(Ex), security software can monitor when certain API calls are used, such as CreateRemoteThread. Combining this information with other data such as process reputation and enterprise-wide telemetry can be used to provide high fidelity alerts for potential malware.

When process injection occurs, one process modifies the memory protections of a memory region in another process's address space. By detecting the use of API calls such as VirtualProtectEx that result in one process modifying the memory protections of address space allowed to another process, especially when the PAGE_EXECUTE_READWRITE permissions are used as this permission is used to allow the shellcode to be written and executed within the same memory space.

As red teamers and malicious actors continue to develop new process injection techniques, network defenders and security software continue to adapt to the ever-changing landscape. Monitoring Windows API function calls such as VirtualAllocEx, VirtualProtectEx, CreateRemoteThread, and NTQueueAPCThread can provide valuable data for identifying potential malware. Monitoring for the use of CreateProcess with the CREATE_SUSPENDED and CREATE_HIDDEN flags may assist in detecting process injection where the attacker creates a suspended and hidden process to inject into.

As we've seen, process injection techniques tend to follow a consistent order in which they call Windows API functions. For example, both injection techniques call VirtualAllocEx followed by WriteProcessMemory and identifying when a process calls these two APIs in that order can be used as a basis for detecting process injection.

Conclusion

Using shellcode as the final stage for payloads during assessments allows Red Teams the flexibility to execute payloads in a wide array of environments while implementing techniques to avoid detection. The DueDLLigence shellcode runner is a dynamic tool that takes advantage of the evasive properties of both shellcode and process injection to offer Red Teams a way to avoid detection. Detections for the execution of LOLbins on the command line and process injection at the API and process level should be incorporated into defensive methodology, as attackers are increasingly being forced into living off the land with the

Posted in Threat Intelligence—Security & Identity

Related articles

Threat Intelligence

Hybrid Russian Espionage and Influence Campaign Aims to Compromise Ukrainian Military Recruits and Deliver Anti-Mobilization Narratives

By Google Threat Intelligence Group • 10-minute read

Threat Intelligence

Investigating FortiManager Zero-Day Exploitation (CVE-2024-47575)

By Mandiant • 19-minute read

Threat Intelligence

How Low Can You Go? An Analysis of 2023 Time-to-Exploit Trends

By Mandiant • 10-minute read

Threat Intelligence

capa Explorer Web: A Web-Based Tool for Program Capability Analysis

By Mandiant • 6-minute read