Introducing Early Cascade Injection: from Windows process creation to stealthy injection

outflank.nl/blog/2024/10/15/introducing-early-cascade-injection-from-windows-process-creation-to-stealthy-injection

Dima van de Wouw October 15, 2024

By Guido Miggelenbrink at Outflank

Introduction

In this blog post we introduce a novel process injection technique named Early Cascade Injection, explore Windows process creation and identify how several Endpoint Detection and Response systems (EDRs) initialize their in-process detection capabilities. This new Early Cascade Injection technique targets the user-mode part of process creation and combines elements of the well-known Early Bird APC Injection technique with the recently published EDR-Preloading technique by Marcus Hutchins [1]. Unlike Early Bird APC Injection, this new technique avoids queuing cross-process Asynchronous Procedure Calls (APCs), while having minimal remote process interaction. This makes Early Cascade Injection a stealthy process injection technique that is effective against top tier EDRs while avoiding detection.

To provide insights into Early Cascade Injection's internals, this blog also presents a timeline of the user-mode process creation flow. This overview illustrates how Early Cascade Injection operates and pinpoints the exact moment at which it intervenes in process creation. Furthermore, we compare that to the initialization timing of EDR user-mode detection measures.

Now, let's dive into the details of Windows process creation, Early Bird APC Injection, and EDR-Preloading. Once we have a solid understanding of these topics, we can proceed to explore Early Cascade Injection.

Understanding Windows process creation

Process creation APIs

In Windows there are various APIs to create a process, such as `CreateProcess`, `CreateProcessAsUser`, and `CreateProcessWithLogon`, as shown in figure 1. Ultimately, all these functions invoke the NAPI `NtCreateUserProcess` in `ntdll.dll`. This function is responsible for initiating process creation by switching control to the kernel, where the equally named function `NtCreateUserProcess` is executed.

Each of these functions include the `dwCreationFlags` parameter, which controls how the process is created. In this post, we'll encounter the `CREATE_SUSPENDED` flag, which instructs the kernel to create the new process's initial thread in a suspended state [2]. The thread remains suspended until the `ResumeThread` function is called.

Obviously, these functies also have a parameter specifying the path to the application's image file for which a process is to be created. Refer to the MSDN for the other parameters and flags of these APIs [2].

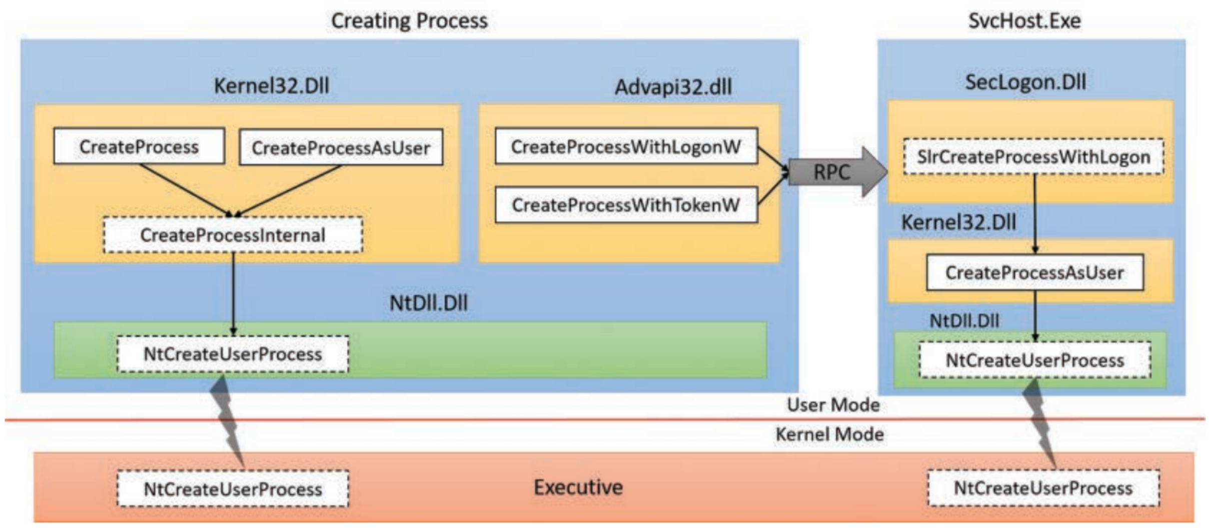


Figure 1: Process creation functions (Source: Windows Internals, Part 1)

Kernel-mode and user-mode process creation

Process creation has two parts: kernel-mode and user-mode. It begins with the kernel-mode part, initiated by `NtCreateUserProcess`. Once the process's context and environment are created in kernel-mode, the initial thread of the process completes process creation in user-mode.

The kernel-mode part is responsible for opening the image file of the specified application and mapping it into memory. It then creates process-specific and thread-specific objects, maps the native library `ntdll.dll` into the process, followed by the creation of the process's initial thread. If the `CREATE_SUSPENDED` flag is specified, this thread is created in suspended state, waiting to be resumed before control returns to user-mode for the remainder of the process creation.

The module `ntdll.dll` is the first DLL loaded into a process and it is the only DLL loaded in kernel-mode, all other modules are loaded in user-mode. Further, `ntdll.dll` includes the exported function `LdrInitializeThunk`, which handles the user-mode part of process creation before the application's main entry point runs. This function is also known as the image loader and the functions related to it in `ntdll.dll` are prefixed with `Ldr`.

Returning to the newly created thread: upon resumption of this suspended thread, it starts executing `LdrInitializeThunk`, the user-mode part of process creation. After, the new process is fully initialized and ready to run the application. The initial thread then begins executing the application's main entry point.

> Note that using the `CREATE_SUSPENDED` flag pauses process creation just before the initial thread switches to user-mode to run `LdrInitializeThunk`. This is particularly interesting because **user-mode malware can interfere with process creation at this point**. Therefore, let's take a closer look at what happens inside `LdrInitializeThunk`.

User-mode process creation: `LdrInitializeThunk`

`LdrInitializeThunk` is the first function executed in user-mode, marking the initial point where malware and EDRs can intervene in a process. We will later explore how techniques such as Early Bird APC Injection, EDR-Preloading and Early Cascade Injection interact with `LdrInitializeThunk`. For now, let's delve into the details of this function.

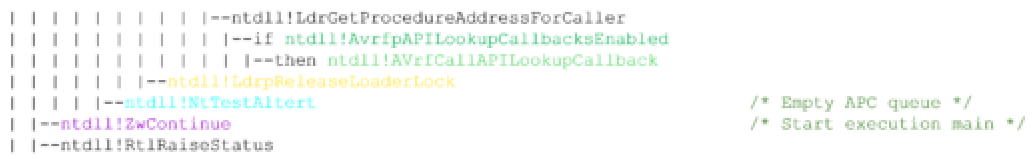
`LdrInitializeThunk` is a complex function responsible for the user-mode part of process creation. It handles numerous process initialisation tasks, which are listed and briefly described in the *Windows Internals, Part 1* book. However, the book does not cover which subordinate functions within `LdrInitializeThunk` are responsible for these tasks. Hence, to gain a deeper understanding of `LdrInitializeThunk` and its subordinate functions, we analysed it using x64dbg and IDA Pro.

Based on this analysis, we created a call graph that outlines the sequence of events in the user-mode part of process creation. This timeline includes the functions relevant to this post and thus omits some tasks and associated functions of `LdrInitializeThunk`. Additionally, please note that this call graph reflects our interpretation and may not be entirely accurate.

The call graph of `LdrInitializeThunk` is depicted below, followed by a description of the tasks that can be recognized in it. Key functions are highlighted in color.

Call graph `LdrInitializeThunk`:

```
|--ntdll!LdrInitializeThunk
| |--ntdll!LdrpInitialize
| | |--ntdll!LdrpInitializeInternal
| | | |--ntdll!_LdrpInitialize
| | | | |--ntdll!LdrInitializeMrdata                                  /* Initialize .mrdata*/
| | | | |--ntdll!LdrpInitializeProcess
| | | | | |--ntdll!LdrpInitializeProcessHeap
| | | | | |--ntdll!LdrpInitParallelLoadingSupport
| | | | | |--ntdll!LdrpCallTlsInitializer
| | | | | | |--ntdll!LdrpCallTlsInitRoutine
| | | | | | | |--ntdll!ImageTlsCallbackCaller
| | | | | |--ntdll!LdrLoadDll : Kernel32DllName                       /* Load kernel32.dll */
| | | | | | |--ntdll!LdrpIntializeDllPath
| | | | | | |--ntdll!LdrpLoadDll
| | | | | | | |--ntdll!LdrpLoadDllInternal
| | | | | | | | |--ntdll!LdrpFindOrPrepareLoadingModule               /* Map module */
| | | | | | | | | |--ntdll!LdrpAllocatePlaceHolder
| | | | | | | | | | |--ntdll!RtlAllocateHeap
| | | | | | | | | | |--ntdll!LdrpAllocateModuleEntry
| | | | | | | | | |--ntdll!LdrpLoadKnownDll
| | | | | | | | | | |--ntdll!LdrpFindKnownDLL
| | | | | | | | | | | |--ntdll!ZwOpenSection
| | | | | | | | | | |--ntdll!LdrpMapDllWithSectionHandle
| | | | | | | | | | | |--ntdll!LdrpMinimalMapModule
| | | | | | | | | | | | |--ntdll!LdrpMapViewOfSection
| | | | | | | | | | | | | |--ntdll!NtMapViewOfSection                 /* Map kernel32.dll */
| | | | | | | | | | |--ntdll!LdrpInsertDataTableEntry
| | | | | | | | | | |--ntdll!LdrpCompleteMapModule
| | | | | | | | | | |--ntdll!LdrpProcessMappedModule
| | | | | | | | | | |--ntdll!LdrpLogNewDllLoad
| | | | | | | | | | |--ntdll!LdrpMapAndSnapDependency                 /* Map dependency */
| | | | | | | | | | | |--ntdll!LdrpPrepareImportAddressTableForSnap
| | | | | | | | | | | |--ntdll!LdrpGetImportDescriptorForSnap
| | | | | | | | | | | |--ntdll!LdrpLoadDependentModule
| | | | | | | | | | | | |--ntdll!LdrpLoadDependentModuleInternal
| | | | | | | | | | | | | |--ntdll!LdrpMapDllWithSectionHandle
| | | | | | | | | | | | | | |--ntdll!LdrpMinimalMapModule
| | | | | | | | | | | | | | | |--ntdll!LdrpMapViewOfSection
| | | | | | | | | | | | | | | | |--ntdll!NtMapViewOfSection           /* Map kernelbase.dll */
| | | | | | | | | | | | | | |--ntdll!LdrpInsertDataTableEntry
| | | | | | | | | | | | | | |--ntdll!LdrpCompleteMapModule
| | | | | | | | | | | | | | |--ntdll!LdrpProcessMappedModule
| | | | | | | | | | | | | | |--ntdll!LdrpLogNewDllLoad
| | | | | | | | | | | | | | |--ntdll!LdrpMapAndSnapDependency         /* Map dependency */
| | | | | | | | | | | | | | | |--ntdll!LdrpPrepareImportAddressTableForSnap
| | | | | | | | | | | | | | | | |--ntdll!LdrpGetImportDescriptorForSnap
| | | | | | | | | | | | | | | | |--ntdll!LdrpLoadDependentModule      /* No other dependency found */
| | | | | | | | |--ntdll!LdrpPrepareModuleForExecution                /* Initialize mapped modules */
| | | | | | | | | |--ntdll!LdrpCondenseGraph
| | | | | | | | | |--ntdll!LdrpNotifyLoadOfGraph
| | | | | | | | | | |--ntdll!LdrpSendPostSnapNotifications
| | | | | | | | | | | |--ntdll!RtlEnterCriticalSection : LdrpDllNotificationLock
| | | | | | | | | | | | |--if ntdll!g_ShimsEnabled
| | | | | | | | | | | | | |--then ntdll!g_pfnSE_DllLoaded
| | | | | | | | | | | |--ntdll!RtlLeaveCriticalSection : LdrpDllNotificationLock
| | | | | | | | | |--ntdll!LdrpAcquireLoaderLock
| | | | | | | | | |--ntdll!LdrpInitializeGraphRecurse
| | | | | | | | | | |--ntdll!LdrpInitializeNode                       /* Initialize kernelbase.dll */
| | | | | | | | | | | |--ntdll!LdrpCallTlsInitalizer
| | | | | | | | | | | |--ntdll!LdrpCallInitRoutine
| | | | | | | | | | | | |--kernelbase!KernelBaseDllIntialize          /* Call entry point kernelbase.dll */
| | | | | | | | | | | | | |--ntdll!LdrGetProcedureAddressForCaller
| | | | | | | | | | | | | | |--if ntdll!AvrfpAPILookupCallbacksEnabled
| | | | | | | | | | | | | | | |--then ntdll!AVrfCallAPILookupCallback
| | | | | | | | | | |--ntdll!LdrpInitializeNode                       /* Initialize kernel32.dll */
| | | | | | | | | | | |--ntdll!LdrpCallTlsInitalizer
| | | | | | | | | | | | |--ntdll!LdrpCallInitRoutine
| | | | | | | | | | | | | |--kernel32!BaseDllInitialze                /* Call entry point kernel32.dll */
| | | | | | | | | |--ntdll!LdrpReleaseLoaderLock
| | | | | |--ntdll!LdrpEnableParallelLoading                          /* Start loading other dependencies */
| | | | | | |--ntdll!LdrpDetectDetour
| | | | | |--ntdll!LdrpMapAndSnapDependency                           /* Map dependencies */
| | | | | | |--ntdll!LdrpLoadDependentModule
| | | | | |--ntdll!LdrpDrainWorkQueue
| | | | | | |--ntdll!pProcessWork
| | | | | |--ntdll!LdrpPrepareModuleForExecution                      /* Initialize mapped modules */
| | | | | | |--ntdll!LdrpCondenseGraph
| | | | | | |--ntdll!LdrpNotifyLoadOfGraph
| | | | | | | |--ntdll!LdrpSendPostSnapNotifications
| | | | | | | | |--ntdll!RtlEnterCriticalSection : LdrpDllNotificationLock
| | | | | | | | | |--if ntdll!g ShimsEnabled
| | | | | | | | | | |--then ntdll!g pfnSE DllLoaded
| | | | | | | | |--ntdll!RtlLeaveCriticalSection : LdrpDllNotificationLock
| | | | | | |--ntdll!LdrpAcquireLoaderLock
| | | | | | |--ntdll!LdrpInitializeGraphRecurse
| | | | | | | |--ntdll!LdrpInitializeNode
| | | | | | | | |--ntdll!LdrpCallTlsInitalizer
| | | | | | | | |--ntdll!LdrpCallInitRoutine
```



```
| | | | | | | | | |--ntdll!LdrGetProcedureAddressForCaller
| | | | | | | | | | |--if ntdll!AvrfpAPILookupCallbacksEnabled
| | | | | | | | | | | |--then ntdll!AVrfCallAPILookupCallback
| | | | | | |--ntdll!LdrpReleaseLoaderLock
| | | | |--ntdll!NtTestAltert                                          /* Empty APC queue */
| |--ntdll!ZwContinue                                                 /* Start execution main */
| |--ntdll!RtlRaiseStatus
```

This call graph illustrates the following tasks performed by `LdrInitializeThunk`:

1. Initialises Loader Lock in the Process Environment Block (`PEB`).
   > The `PEB` is a data structure that stores information about the process's context and environment. Loader Lock will be discussed later.
2. Sets up the Mutable Read-Only Heap Section (.mrdata).
3. Creates and inserts the first loader data table entry (`LDR_DATA_TABLE_ENTRY`) for `ntdll.dll` into the module database (`PEB_LDR_DATA`).
   > The module database, stored in the `PEB`, contains three lists that keep track of the process's loaded modules: `InLoadOrderModuleList`, `InMemoryOrderModuleList`, and `InInitializationOrderModuleList`. Each entry in these lists includes details such as the module's base address, entry point, and path.
4. Initialises the parallel loader.
   > The parallel loader is responsible for loading the application's imported DLLs concurrently, using a pool of threads. It sets up a `LdrpWorkQueue` that contains the application's first order dependencies to be loaded. Then, the parallel threads load these dependencies, recursive ones are added to the work queue. For more information on the Windows parallel loader, refer to this insightful blog by [3].
5. Creates and inserts the second loader data table entry (`LDR_DATA_TABLE_ENTRY`) for the application's executable into the module database (`PEB_LDR_DATA`).
6. Loads the initial modules `kernel32.dll` and `kernelbase.dll`.
   > These modules are always loaded into every process. Like all other modules, these are also added to the module database (`PEB_LDR_DATA`).
7. If enabled, initializes the Shim Engine and parses the Shim Database. **The Shim Engine is by default off.**
   > The Shim Engine applies compatibility fixes (shims) to applications without modifying their code. It intercepts and modifies API calls to address compatibility issues. For more details, refer to [11].
8. If enabled, it uses the parallel loader to load the application's remaining dependencies into the process; otherwise, DLLs are loaded sequentially.

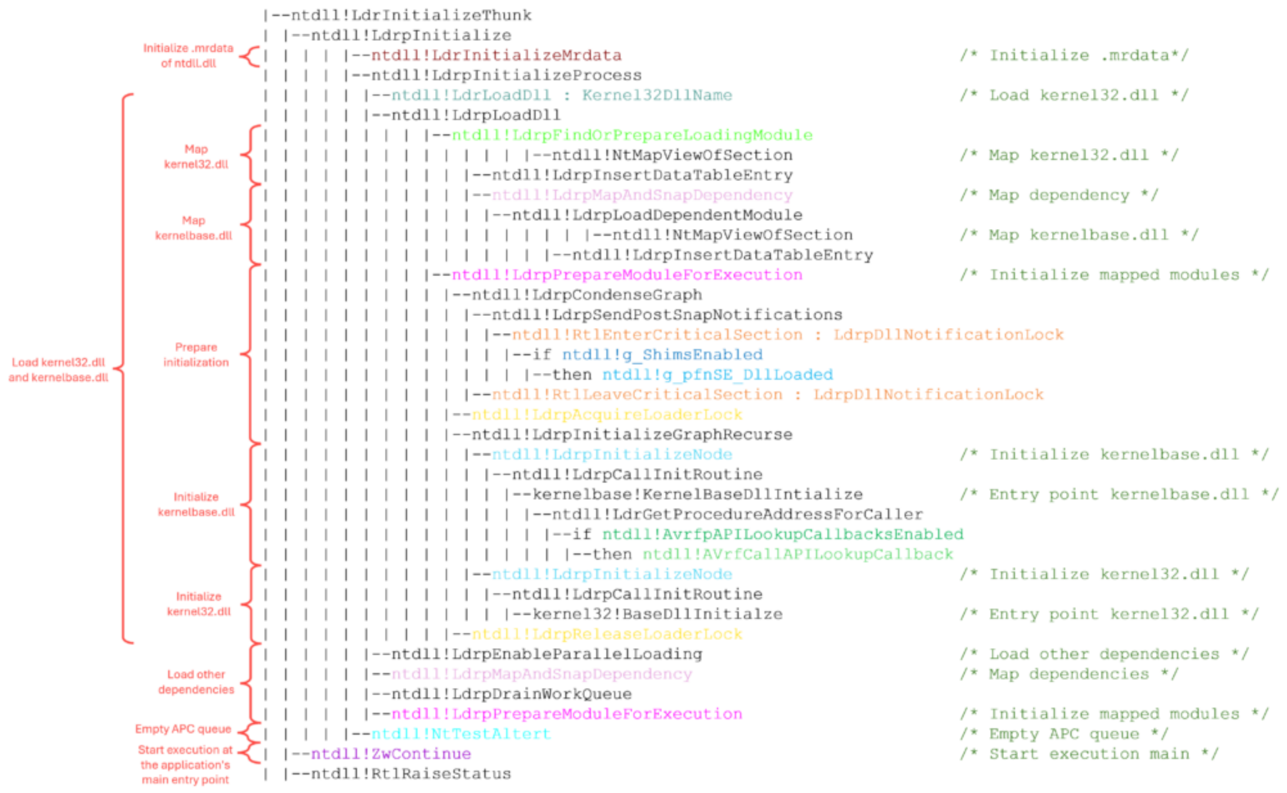
After mapping and initializing all dependencies, `LdrInitializeThunk` invokes the following functions:

9. `NtTestAlert`: **Empties the APC queue of the calling thread** by switching to kernel-mode, which then calls `KiUserApcDispatcher` to execute the queued APCs.
10. `NtContinue`: Sets the initial thread's execution context to `RtlUserThreadStart`, which subsequently invokes the main entry point of the application.

At the very bottom of the call graph, we see `RtlRaiseStatus`. This function is normally never executed, as `NtContinue` redirects the execution flow to the application's main entry point. However, if the application crashes, `RtlRaiseStatus` is triggered.

Although the call graph is simplified, it remains complex. Hopefully, it provides some insight into the internals of process creation. To clarify the key points relevant to this blog, we created an abstraction that captures the essential information you need to remember. The red comments describe the operation that the preceding block of functions accomplish.

Abstracted call graph `LdrInitializeThunk`:



Initialize .mrdata of ntdll.dll
Load kernel32.dll and kernelbase.dll
Map kernel32.dll
Map kernelbase.dll
Prepare initialization
Initialize kernelbase.dll
Initialize kernel32.dll
Load other dependencies
Empty APC queue
Start execution at the application's main entry point

```
|--ntdll!LdrInitializeThunk
| |--ntdll!LdrpInitialize
| | | | |--ntdll!LdrInitializeMrdata                                          /* Initialize .mrdata*/
| | | | |--ntdll!LdrpInitializeProcess
| | | | | |--ntdll!LdrLoadDll : Kernel32DllName                                /* Load kernel32.dll */
| | | | | |--ntdll!LdrpLoadDll
| | | | | | | | |--ntdll!LdrpFindOrPrepareLoadingModule
| | | | | | | | | | | | | |--ntdll!NtMapViewOfSection                         /* Map kernel32.dll */
| | | | | | | | | | |--ntdll!LdrpInsertDataTableEntry
| | | | | | | | | | |--ntdll!LdrpMapAndSnapDependency                          /* Map dependency */
| | | | | | | | | | | |--ntdll!LdrpLoadDependentModule
| | | | | | | | | | | | | | | | |--ntdll!NtMapViewOfSection                   /* Map kernelbase.dll */
| | | | | | | | | | | | | | |--ntdll!LdrpInsertDataTableEntry
| | | | | | | | |--ntdll!LdrpPrepareModuleForExecution                          /* Initialize mapped modules */
| | | | | | | | | |--ntdll!LdrpCondenseGraph
| | | | | | | | | | |--ntdll!LdrpSendPostSnapNotifications
| | | | | | | | | | | |--ntdll!RtlEnterCriticalSection : LdrpDllNotificationLock
| | | | | | | | | | | | |--if ntdll!g_ShimsEnabled
| | | | | | | | | | | | | |--then ntdll!g_pfnSE_DllLoaded
| | | | | | | | | | |--ntdll!RtlLeaveCriticalSection : LdrpDllNotificationLock
| | | | | | | | | |--ntdll!LdrpAcquireLoaderLock
| | | | | | | | | |--ntdll!LdrpInitializeGraphRecurse
| | | | | | | | | | |--ntdll!LdrpInitializeNode                                /* Initialize kernelbase.dll */
| | | | | | | | | | | |--ntdll!LdrpCallInitRoutine
| | | | | | | | | | | | |--kernelbase!KernelBaseDllIntialize                 /* Entry point kernelbase.dll */
| | | | | | | | | | | | | |--ntdll!LdrGetProcedureAddressForCaller
| | | | | | | | | | | | | | |--if ntdll!AvrfpAPILookupCallbacksEnabled
| | | | | | | | | | | | | | | |--then ntdll!AVrfCallAPILookupCallback
| | | | | | | | | | |--ntdll!LdrpInitializeNode                                /* Initialize kernel32.dll */
| | | | | | | | | | | |--ntdll!LdrpCallInitRoutine
| | | | | | | | | | | | |--kernel32!BaseDllInitialze                         /* Entry point kernel32.dll */
| | | | | | | | | |--ntdll!LdrpReleaseLoaderLock
| | | | | |--ntdll!LdrpEnableParallelLoading                                    /* Load other dependencies */
| | | | | |--ntdll!LdrpMapAndSnapDependency                                    /* Map dependencies */
| | | | | |--ntdll!LdrpDrainWorkQueue
| | | | | |--ntdll!LdrpPrepareModuleForExecution                               /* Initialize mapped modules */
| | | | |--ntdll!NtTestAltert                                                  /* Empty APC queue */
| |--ntdll!ZwContinue                                                          /* Start execution main */
| |--ntdll!RtlRaiseStatus
```

> In this abstraction of the call graph, we observe that `LdrInitializeThunk` loads `kernel32.dll` and `kernelbase.dll`, then it loads all other dependencies, clears its APC queue (`NtTestAlert`), and finally begins executing the application's main entry point (`NtContinue`). Additionally, we observe that loading a DLL via `LdrLoadDll` consists of two steps: mapping and initialization.

How `LdrLoadDll` works

Loading dependencies is a major component of `LdrInitializeThunk`, which is probably why it is called the image loader. Moreover, the techniques EDR-Preloading and Early Cascade Injection specifically intervene in this `LdrLoadDll` function during process creation. Therefore, we also briefly cover how `LdrLoadDll` loads dependencies. After that, we will start exploring the interesting stuff: process injection.

> The most important aspect to remember after this section is that`LdrLoadDll` loads a DLL in two steps: first it **maps** the DLL's image file into memory through `LdrpFindOrPrepareLoadingModule` and second it **initializes** the DLL via `LdrpPrepareModuleForExecution`. If a DLL has recursive dependencies, these are first mapped into memory and only then each module is initialized. Initialization is performed in reverse order to follow the correct dependency sequence.

Now, let's delve into a bit more details and walk through the process of loading `kernel32.dll`, the first module being loaded during process creation. Try to follow along in the call graph. We start at `LdrInitializeThunk` which calls `LdrpLoadDll` to load `kernel32.dll`. `LdrpLoadDll` invokes `ntdll!LdrpFindOrPrepareLoadingModule` for the first step: mapping `kernel32.dll` into memory. The actual mapping is eventually performed by the nested function `NtMapViewOfSection`. After, `LdrpMapAndSnapDependency` checks for dependencies, and since `kernel32.dll` imports functions from `kernelbase.dll`, `LdrpLoadDependentModule` maps `kernelbase.dll` into memory.

Once the two modules have been mapped into memory, the second part of `LdrLoadDll` initializes the DLLs, carried out by `LdrpPrepareModuleForExecution`. This function invokes `LdrpCondenseGraph` to create a dependency graph, which stores the order in which dependencies must be initialized. Following, `LdrpInitializeGraphRecurse` processes this graph and for each module in the graph `LdrpInitializeNode` is called. The first node in the graph is `kernelbase.dll` which `LdrpInitializeNode` initialises through `LdrpCallInitRoutine`. This function calls the entry point of `kernelbase.dll`. After that, the same is performed to initialize `kernel32.dll`; hereafter, `LdrLoadDll` has finished loading `kernel32.dll`.

Early Bird APC Injection

Now that we have a general understanding of process creation, let's take a closer look at the Early Bird APC Injection technique which was discovered by Cyberbit in 2018 [4]. This is a well-known and effective process injection method that involves injecting code before the execution of a process's main entry point. Injecting this early in the process may evade EDR detection measures, including hooks, if these measures are not loaded before APC execution.

The Early Bird APC Injection technique works as follows:

1. Create a target process in suspended state (e.g. `CreateProcess`);
2. Allocate writeable memory in the target process (e.g. `VirtualAllocEx`);
3. Write malicious code to the allocated memory (e.g. `WriteProcessMemory`);
4. Queue an APC to the remote target process, the APC points to the malicious code (e.g. `QueueUserAPC`);

5. Resume the target process, upon resumption the APC is executed, running the malicious code (e.g. `ResumeThread`).

As we previously learned, when a process is created in a suspended state (1), execution halts just before the user-mode part of process creation, handled by `LdrInitializeThunk`. At this point, the payload with malicious code is written into the target process (2, 3). Then, an APC routine pointing to the payload is queued for execution in the suspended thread (4). Last, the suspended thread is resumed (5).

On resumption of the thread, it starts execution at `LdrInitializeThunk` and one of final tasks of `LdrInitializeThunk` is emptying the APC queue. Specifically, `NtTestAlert` is responsible for emptying a threads APC queue by executing the APCs in it. This is when the injected payload runs.

In the past, execution of the payload at this point was early enough to preempt EDR user-mode detection measures, like hooks. However, modern EDR solutions often load their detection measures earlier in the process creation timeline. Nonetheless, we found that a popular EDR still loads its detection measures after `NtTestAlert`. For this particular EDR, Early Bird APC Injection bypasses the EDR's user-mode detection measures. Despite Early Bird APC Injection does may no longer evade hooks of modern EDRs, it still remains very useful for injection purposes.

> Early Bird APC remains a valuable injection technique, even though it is less effective as an evasion method against modern EDRs.

Nevertheless, as mentioned earlier, **Early Bird APC Injection is likely to be detected due to the suspicious cross-process queuing of an APC**. Queuing an APC from one process to another is known as cross-process APC queueing. This behaviour is suspicious and closely monitored by EDRs. It's difficult to hide cross-process APC queueing, making it a strong indicator for detecting Early Bird APC Injection. In a moment, we will see how Early Cascade Injection performs its injection without cross-process queuing and therefore goes undetected against the EDRs we tested.

EDR-Preloading

Early Cascade Injection incorporates elements of EDR-Preloading. Therefore, let's briefly delve into EDR-Preloading, which was recently introduced in a blog by Marcus Hutchins, who is known for stopping the WannaCry [1]. His blog inspired my research in this area, thank you for that!

EDR-Preloading is designed to prevent EDRs from loading their user-mode detection measures during process creation. For example, it prevents EDRs from initialising their Hooking DLL, which significantly reduces an EDRs the visibility within the process as the EDR is unable to intercept API calls. Techniques like this are becoming increasingly important as Microsoft gradually restricts third-party access to the kernel, forcing EDR detection measures from kernel-mode to user-mode. It is speculated that kernel restrictions will be pushed further since the Crowdstrike incident took down 8.5 million Windows systems, due to a faulty update in their kernel-level software [5].

How EDR-Preloading works: **it begins by creating a process in suspended state and hijacking the `ntdll!AvrfpAPILookupCallbackRoutine` callback pointer in `ntdll.dll`**. Hijacking involves assigning the start address of malicious code to the callback pointer and enabling the callback pointer by setting `ntdll!AvrfpAPILookupCallbacksEnabled` to `1`. As a result, the callback pointer is executed during the user-mode part of process creation after resuming the suspended process. **Once invoked, the malicious code runs, thereby taking control of the execution flow during process creation.**

This malicious code runs very early in the process creation sequence when only `ntdll.dll` is loaded. Specifically, the callback `AvrfpAPILookupCallbackRoutine` is triggered in the initialisation part of `LdrLoadDll`, as illustrated in the call graph. Both the callback pointer (`ntdll!AvrfpAPILookupCallbackRoutine`) and the boolean variable (`ntdll!AvrfpAPILookupCallbacksEnabled`) are highlighted in the graph in light green and green. This `LdrLoadDll` function is executed for the first time during the initialization of `kernelbase.dll`, the first DLL to be loaded in user-mode. If EDRs load their detection measures after this point, it is possible to prevent them from loading their detection measures. A detailed explanation of this and how to implement it can be found in the EDR-Preloading blog.

> What we found interesting about the EDR-Preloading technique is that it possible to get code execution just by overwriting a callback pointer in the target's `ntdll.dll` during process creation. **However, this code execution is highly constrained.**

Code execution limitations

The code execution obtained through `ntdll!AvrfpAPILookupCallbackRoutine` during process initialisation is significantly constrained. These limitations are caused by the **limited number of available dependencies** and the constraints imposed by the **Loader Lock** synchronisation object.

Limited dependencies

At the point when the `AvrfpAPILookupCallbackRoutine` callback is invoked, only `ntdll.dll` is fully loaded into the process. Consequently, code execution is restricted to the undocumented NTAPI functions within `ntdll.dll`, significantly limiting the actions that can be performed. The lack of access to other libraries, such as `winhttp.dll`, complicates the execution of more complex operations, such as communicating with a command and control (C2) server.

Furthermore, due to the presence of Loader Lock, no additional DLLs can be loaded, and no new threads can be created.

Loader Lock

The `ntdll!AvrfpAPILookupCallbackRoutine` callback runs during the initialisation part of `LdrLoadDll`, under the function `LdrpPrepareModuleForExecution`. More precisely, the callback is triggered within `LdrpInitializeNode`, which handles the actual initialisation of a DLL, as previously discussed. During the execution of `LdrpInitializeNode`, Loader Lock is held to synchronize the loading and unloading of DLLs. The call graph shows that this synchronization is managed by `LdrpAcquireLoaderLock` and released by `LdrpReleaseLoaderLock`.

Loader Lock is a critical section object that prevents the loading of additional DLLs and creation of new threads [8]. Critical sections are a synchronisation mechanism similar to mutexes and semaphores, but they are designed to be more efficient and for single-process synchronisation. For information on critical section objects, refer to the MSDN documentation [8].

Loader Lock is acquired each time when a function needs access to the module database (`PEB_LDR_DATA`), which is involved in tasks such as DLL loading, unloading, and thread creation [9]. We discussed the module database earlier in step 3 of `LdrInitializeThunk`'s tasks. A well-known function that accesses the module database is `GetModuleHandle`, which retrieves the base address of a DLL and is often used by malware to resolve undocumented NTAPI functions. However, if this function is called while Loader Lock is active, such as during the execution of `AvrfpAPILookupCallbackRoutine`, a deadlock occurs, causing the process to hang. Similarly, attempting to load additional DLLs via functions like `LdrLoadDll` results in a deadlock.

> In summary, code execution through the callback pointer `AvrfpAPILookupCallbackRoutine` during process initialisation is limited to the modules already loaded into the process at that point (`ntdll.dll`). Additionally, Loader Lock prevents the loading of additional DLLs and the creation of new threads, making it difficult to execute tasks that require access to more modules. Despite these limitations, the EDR-Preloading technique has demonstrated that there are just enough capabilities to prevent EDRs from loading their detection measures.

Early Cascade Injection: A new process injection technique

What we found interesting about the EDR-Preloading technique is that you can get code execution just by overwriting a callback pointer in the target's `ntdll.dll` during process creation. However, as we have seen, the code execution obtained through this callback is highly restricted as Loader lock is enabled, making it impractical to run fully functional code, such as an implant with networking capabilities. Therefore, we explored novel and alternative techniques during process creation for process injection. As a results we developed Early Cascade Injection, a novel code injection technique emerged from the restrictions imposed by the Loader Lock.

An alternative callback pointer: `g_pfnSE_DllLoaded`

During our search for alternative injection techniques, **we discovered an alternative callback pointer that also allows code execution during the user-mode part of process creation.** This pointer, named `g_pfnSE_DllLoaded`, is located in the `.mrdata` section of `ntdll.dll`. Unlike the `AvrfpAPILookupCallbackRoutine`, `g_pfnSE_DllLoaded` does not appear to run under Loader Lock. This can be inferred from the call graph, where it is highlighted in light blue and blue.

While not directly relevant to this blog, it might be interesting to understand what this pointer belongs to. The `g_pfnSE_DllLoaded` pointer belongs to the Shim Engine, as indicated by its name, the prefix `g_pfnSE` stands for “global function pointer Shim Engine”. The Shim Engine is a Windows technology responsible for applying compatibility fixes, known as ‘shims,’ without modifying application code. It allows older applications to run on newer versions of Windows, by intercepting and modifying API calls. Although rarely used and disabled by default, the Shim Engine’s implementation is still present in `ntdll.dll`, along with its pointers, including `g_pfnSE_DllLoaded`.

Let’s return to the key aspects of `g_pfnSE_DllLoaded`. **The pointer can be manually enabled by setting the `g_ShimsEnabled` boolean variable to `1`**, located in the `.data` section of `ntdll.dll`. **However, enabling this variable enables all Shim Engine related pointers**, not just `g_pfnSE_DllLoaded`. Each of these pointers require a valid address, and if any remain uninitialized, the process will crash. This makes it impractical to exploit `g_pfnSE_DllLoaded` alone without addressing the other pointers.

To overcome this, **we focused specifically on `g_pfnSE_DllLoaded` as it is the first Shim Engine pointer invoked during process creation.** By targeting this pointer we can execute code before any of the other unassigned pointers and prevent them from executing. This method involves assigning the address of our shellcode to `g_pfnSE_DllLoaded` and enabling `g_ShimsEnabled` to activate it. **Upon execution, the shellcode immediately disables `g_ShimsEnabled`, preventing the remaining Shim Engine pointers from being invoked.** This approach allows use to execute code without causing the process to crash due to uninitialized pointers.

Returning to the call graph, we observe that `g_pfnSE_DllLoaded` runs in the scope of `LdrpSendPostSnapNotifications`, which is a subordinate function of `LdrpPrepareModuleForExecution`. Unlike `LdrpPrepareModuleForExecution`, we observe that `g_pfnSE_DllLoaded` does not run under Loader Lock. Instead, a different critical section object is acquired: `LdrpDllNotificationLock`. This critical section appears to be self-reentrant, suggesting it should not lead to deadlock when loading additional DLLs, although we have not verified.

Despite not operating under Loader Lock, we were unable to run a fully functional shellcode. This is likely due to interrupting the loading process of kernelbase.dll and kernel32.dll. We will work around this in the next section.

Let’s briefly revisit the memory section where `g_pfnSE_DllLoaded` resides, as this is crucial for leveraging it. **`g_pfnSE_DllLoaded` is located in the `.mrdata` section, which is writable when a process is created in a suspended state.** Later, during the user-mode part of process initialization, this section is made read-only, as noted in step 2 of `LdrInitializeThunk`. After this step, modifying its content requires memory protection changes.

Furthermore, **the `g_ShimsEnabled` boolean is located in the `.data` section, which remains writable throughout the entire process.** This allows us to enable or disable the `g_pfnSE_DllLoaded` pointer without modifying memory protections. In contrast, the `AvrfpAPILookupCallbacksEnabled` boolean, used in EDR-Preloading, resides in the `.mrdata` section and requires memory protection changes after step 2 of `LdrInitializeThunk`.

This makes `g_pfnSE_DllLoaded` preferable over `AvrfpAPILookupCallbackRoutine`, as it can be disabled without altering memory protections. As a result, the shellcode required to hijack the pointer is smaller, invoked only once, involves fewer API calls, and therefore reduces the risk of detection.

Additionally, **the `g_pfnSE_DllLoaded` pointer is triggered slightly earlier than `AvrfpAPILookupCallbackRoutine`, offering earlier control over the process.** Similar to how `AvrfpAPILookupCallbackRoutine` is leveraged in EDR-Preloading to preempt EDRs, `g_pfnSE_DllLoaded` can also be used for this purpose, with potentially greater effectiveness due to its earlier execution. As shown in the call graph, `g_pfnSE_DllLoaded` is executed just before `LdrpCallInitRoutine`, which initializes a DLL. This timing allows us to disrupt the initialisation of EDR user-mode detection measures implemented as DLL, making them ineffective. For example, it could prevent EDRs from deploying hooks that intercept API calls, significantly reducing an EDRs visibility within a process. While not the focus of this blog, this presents another use case for the pointer.

> In summary, we identified an alternative pointer named `g_pfnSE_DllLoaded`, located in the `.mrdata` section of `ntdll.dll`. This pointer can be enabled via the `g_ShimsEnabled` boolean, located in the `.data` section of `ntdll.dll`. The `.mrdata` section is writable during suspend state of process creation and the `.data` section is writable through the entire process, allowing use to hijack this pointer without changing memory protections. Moreover, `g_pfnSE_DllLoaded` does not operate under Loader Lock, but it is not trivial to execute fully functional shellcode for a unknown reason. Though, we suspect this may be related to a critical section object or because of the interruption during the `kernel32.dll` and `kernelbase.dll` loading process.

Intra-process APC queueing

The limitations of code execution through `g_pfnSE_DllLoaded` made us thinking. We then realized that during the code execution, **we could invoke an execution primitive to run code at a different stage, free from the limitations**. We considered several execution primitives, including `NtQueueApcThread`, `NtCreateThread` and various callbacks such as `CreateTimerQueueTimer`. Eventually, we found that `NtQueueApcThread` was suitable for our needs and did the job [6]. A comprehensive list of potential callbacks as alternative to `NtQueueApcThread` can be found in this repository [7].

The use of an execution primitive to move code execution to another point, e.g. via `NtQueueApcThread`, was inspired by Early Bird APC Injection. Although, Early Bird APC Injection leverages the APC queue for cross-process code execution.

By leveraging the code execution obtained through `g_pfnSE_DllLoaded`, we can have the initial thread queue an APC on itself. This allows us to transition to unrestricted execution later in the process creation. We refer to this as intra-process APC queuing. The queued APC routine points to malicious code in the target's memory, such as an implant.

`NtQueueApcThread` was particular suitable because it is available in `ntdll.dll` and is not subject to the loader lock since it does not involve DLL operations or thread creation. This means we don't have to worry about causing a deadlock when calling this function within the execution scope of `g_pfnSE_DllLoaded`.

Moreover, `NtQueueApcThread` allows us to queue an APC early in the process initialization phase, before the APC queue is emptied. As detailed in step 9 of `LdrInitializeThunk`, one of the final steps involves invoking `NtTestAlert` to clear the APC queue. **This guarantees the execution of our queued APC.** Furthermore, since `NtTestAlert` is one of the final functions, we can be certain that all DLLs, including `kernel32.dll` and `kernelbase.dll`, have been fully loaded, ensuring that no issues arise from incomplete DLL loading.

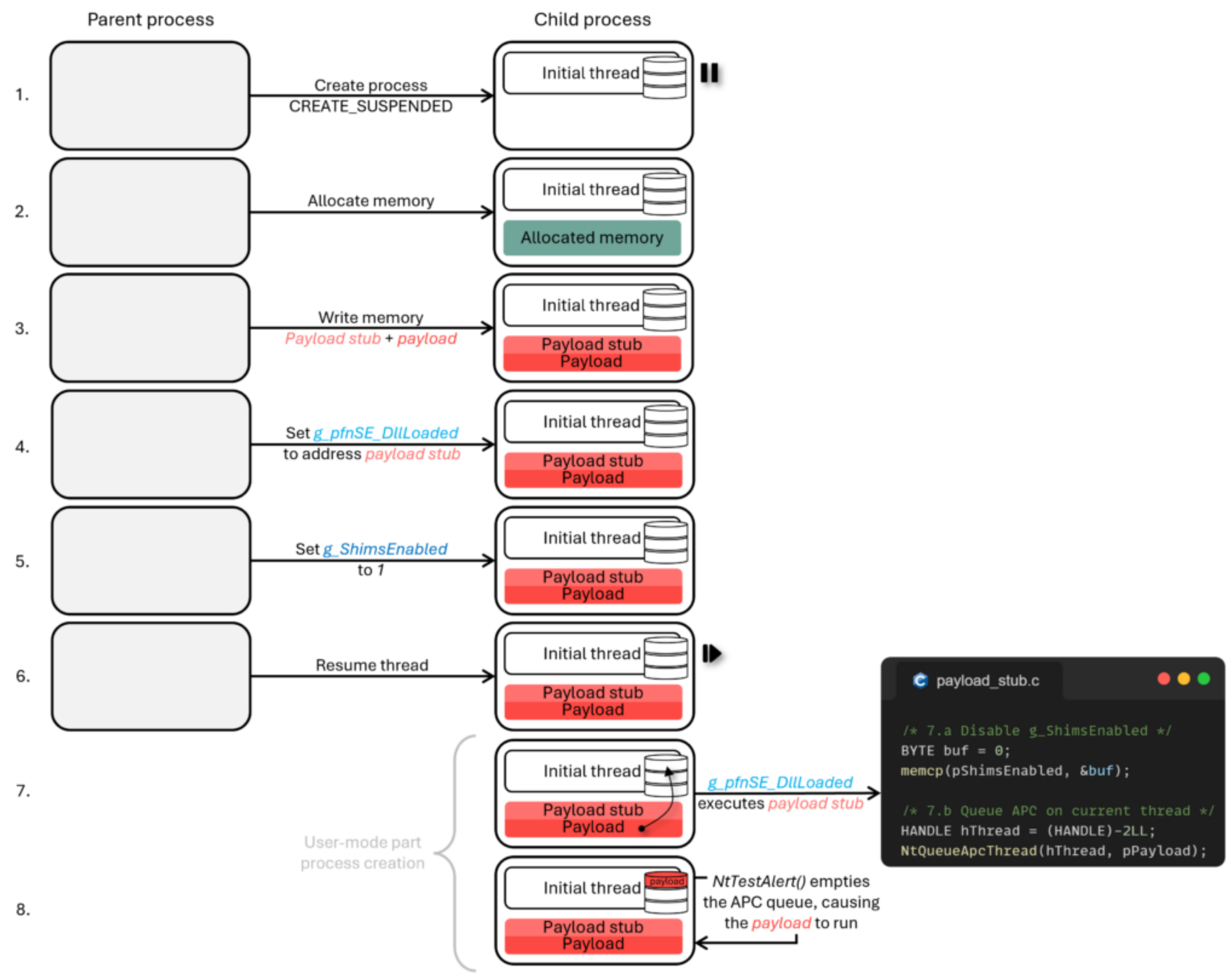
To test our idea, we wrote a piece of shellcode that utilizes `NtQueueApcThread` in `ntdll.dll` to queue an intra-process APC. We refer to this shellcode as the **payload stub**, which we into the target's memory. The APC routine passed to `NtQueueApcThread` points to the address of malicious code that we have written into the target's memory. This malicious code we refer to as the **payload**. Thus, the payload stub is executed by `g_pfnSE_DllLoaded`, while the payload is executed through the APC.

The Early Cascade Injection technique

By now, the direction of the Early Cascade Injection technique should be clear, as we've covered all the key elements and background information. It's time to bring everything together and formally introduce Early Cascade Injection!

Early Cascade Injection works as following: it begins by creating a child process in suspended state. Then it writes a two-part payload into it. Next, the parent locates the pointer `g_pfnSE_DllLoaded` in the `.mrdata` section and it locates the `g_ShimsEnabled` boolean variable in the `.data` section of `ntdll.dll`. Following, it assigns the address of the first payload part, the payload stub, to this `g_pfnSE_DllLoaded` pointer of the new process and enables it by setting `g_ShimsEnabled` to `1`. Last, it resumes the suspended process. As a result, the initial thread of the new process executes the payload stub. This payload immediately disables the `g_ShimsEnabled` by setting it to `0`, preventing the remaining Shim Engine related pointers from executing. Then, the payload stub queues the second part of the payload as an APC on itself, i.e. the initial thread, using `NtQueueApcThread`. This APC is triggered near the end of the Windows Image Loader by the `NtTestAlert` function. As a result, the main payload executes. The main payload could be an implant containing the main functionality that the attackers want to run on the target's system.

In figure 2, the flow of Early Cascade Injection is depicted as described above.



```c
/* 7.a Disable g_ShimsEnabled */
BYTE buf = 0;
memcp(pShimsEnabled, &buf);

/* 7.b Queue APC on current thread */
HANDLE hThread = (HANDLE)-2LL;
NtQueueApcThread(hThread, pPayload);
```

Figure 2: Flow Early Cascade Injection

Early Cascade Injection is a novel injection technique and may serve as alternative to Early Bird APC Injection. The major advantage compared to Early Bird APC Injection is that Early Cascade does not involve a remote execution primitive (cross-process APC queueing). In addition, Early Cascade Injection, unlike Early Bird APC Injection, is currently undocumented and breaks traditional code injection patterns by not queuing an APC across processes. We tested it against multiple EDRs, including top tier ones, and went undetected.

Key features

- **No remote execution primitive:** Early Cascade Injection avoids remote execution primitives such as `QueueUserAPC`. Just like threadless injection methods, it leverages a pointer for execution of the payload, avoiding the need for a remote execution primitive.
- **Minimal remote process interaction:** Early Cascade Injection only involves remote memory allocation, protection and writing.

- **Writable `.mrdata` and `.data`:** The `.mrdata` section is writable during the suspended state, allowing modifications without changing memory protections. Also `.data` is writable during the entire process, allowing enabling/disabling of `g_pfnSE_DllLoaded` without changing memory protections.
- **Novel technique:** Due to Early Cascade Injection's novel approach, its call pattern is less likely to be recognized by security products, reducing the risk of detection.
- **Undocumented callback:** Early Cascade Injection relies on the undocumented pointer `g_pfnSE_DllLoaded`, which may change with Windows updates, potentially impacting its reliability.

EDR detection measure loading mechanism and timing

In this final section, we explore how and when EDRs load their user-mode detection measures, such as hooks, during process creation. Understanding the timing of these measures is crucial for developing strategies to preempt and evade them. Preempting means gaining control of the process before these detection measures are in place. For confidentiality, we won't mention specific EDR names.

To provide a clearer understanding of user-mode detection mechanisms, we will briefly discuss hooks, using them as an example. Besides, user-mode hooks are one of the key detection measures used by EDRs to detect malicious activity. Especially, since Microsoft gradually restricts kernel access, which forces EDRs to shift detection measures to user-mode [5]. Microsoft does offer alternatives like Event Tracing for Windows (ETW), however these are not yet widely adopted. This is likely to change in the nearby future.

Hooks allow EDRs to monitor processes in real time by intercepting API calls from within the process. By preventing these hooks from loading, attackers can significantly reduce an EDR's visibility, thereby increasing the chances of malware evading detection. One effective approach to avoid hooks is by acting before the hooks are fully loaded and take effect. Typically, EDRs place hooks through a hooking DLL during the user-mode part of process creation. In the following section, we will explain how this works in detail.

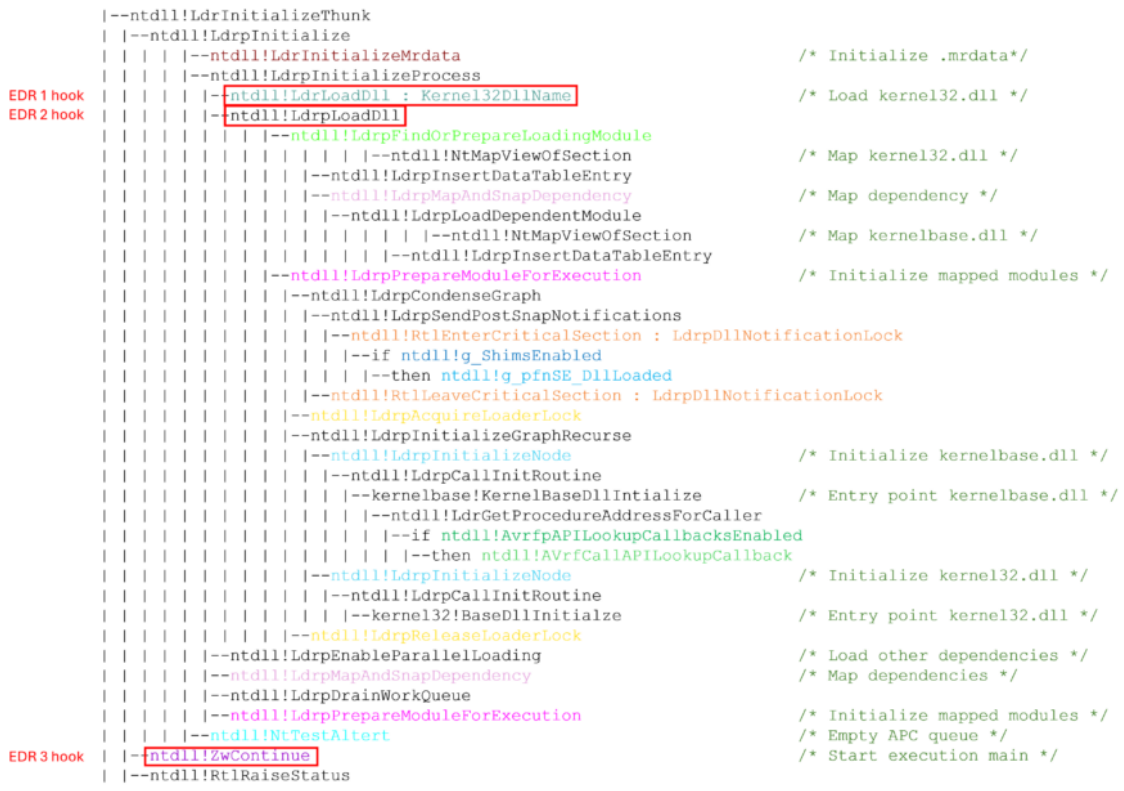
Before diving into the technical details, it's essential to understand the role of the kernel driver in EDRs. This driver enables EDRs to register notification callback routines to receive alerts on system events such as process creation or termination, image loading, registry changes, and system shutdown requests. These callbacks gather system information on which EDRs might take future actions. For instace, upon receiving a process creation notification, the EDR can inject its hooking DLL into a new process for monitoring purposes.

The process notification callback is stored in the kernel's `nt!PspCreateProcessNotifyRoutine` array, which holds all registered callbacks. When a new process is created, the kernel function `nt!PspCallProcessNotifyRoutines` iterates through this array, invoking each callback. For more information on the components of EDRs and their interaction with Windows, we recommend the book Evading EDR by Matt Hand.

As a side note, there are evasion tools that can deregister kernel callbacks to prevent EDRs from loading additional security measures [10]. However, this approach requires access to the kernel, which is typically achieved through the exploitation of a vulnerable kernel driver. By modifying the kernel's notification callbacks, these tools can block the EDR from loading its user-mode detection measures. However, the required kernel-access for this technique, makes it a complex method for evasion.

Returning to the main point, EDRs use the process creation notification as a trigger to load user-mode detection measures into newly created processes. We analyzed several EDRs to understand how these detection measures are loaded. Based on our findings, we explain the general approach EDRs take to inject their user-mode detection modules.

We observed that when a newly created process resumes from a suspended state, EDRs modify `ntdll.dll` just before transitioning from kernel to user-mode (`LdrInitializeThunk`). Specifically, EDRs inject shellcode into the process memory, which contains the logic to load the EDR's hooking DLL. Additionally, they place a hook in `LdrInitializeThunk`, redirecting code execution to the injected shellcode. In our analyses of various EDRs, we found that the hooks are specifically placed on `LdrLoadDll`, `LdrpLoadDll`, or `NtContinue` within `LdrInitializeThunk`. Figure 3 revisits the call graph and highlights these functions. Note, EDRs also load their detection measures using this mechanism for processes that are not created in a suspended state.

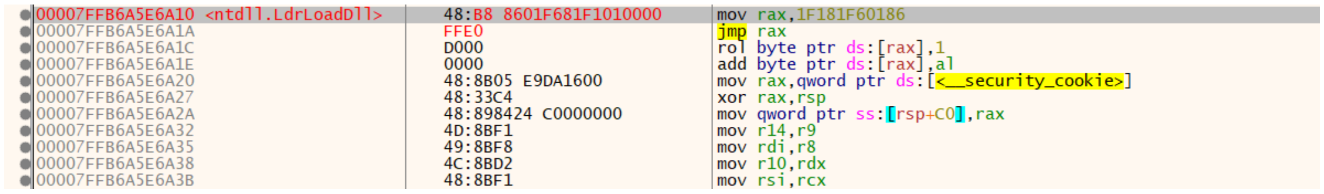


```
             |--ntdll!LdrInitializeThunk
             | |--ntdll!LdrpInitialize
             | | | | |--ntdll!LdrInitializeMrdata                                 /* Initialize .mrdata*/
             | | | | |--ntdll!LdrpInitializeProcess
EDR 1 hook   | | | | | |--ntdll!LdrLoadDll : Kernel32DllName                      /* Load kernel32.dll */
EDR 2 hook   | | | | | |--ntdll!LdrpLoadDll
             | | | | | | | | |--ntdll!LdrpFindOrPrepareLoadingModule
             | | | | | | | | | | | | |--ntdll!NtMapViewOfSection                  /* Map kernel32.dll */
             | | | | | | | | | | |--ntdll!LdrpInsertDataTableEntry
             | | | | | | | | | | |--ntdll!LdrpMapAndSnapDependency                 /* Map dependency */
             | | | | | | | | | | | |--ntdll!LdrpLoadDependentModule
             | | | | | | | | | | | | | | | | |--ntdll!NtMapViewOfSection          /* Map kernelbase.dll */
             | | | | | | | | | | | | | | | |--ntdll!LdrpInsertDataTableEntry
             | | | | | | | | |--ntdll!LdrpPrepareModuleForExecution                 /* Initialize mapped modules */
             | | | | | | | | | |--ntdll!LdrpCondenseGraph
             | | | | | | | | | | |--ntdll!LdrpSendPostSnapNotifications
             | | | | | | | | | | | |--ntdll!RtlEnterCriticalSection : LdrpDllNotificationLock
             | | | | | | | | | | | | |--if ntdll!g_ShimsEnabled
             | | | | | | | | | | | | | |--then ntdll!g_pfnSE_DllLoaded
             | | | | | | | | | | |--ntdll!RtlLeaveCriticalSection : LdrpDllNotificationLock
             | | | | | | | | | |--ntdll!LdrpAcquireLoaderLock
             | | | | | | | | | |--ntdll!LdrpInitializeGraphRecurse
             | | | | | | | | | | |--ntdll!LdrpInitializeNode                      /* Initialize kernelbase.dll */
             | | | | | | | | | | | |--ntdll!LdrpCallInitRoutine
             | | | | | | | | | | | | |--kernelbase!KernelBaseDllIntialize         /* Entry point kernelbase.dll */
             | | | | | | | | | | | | | |--ntdll!LdrGetProcedureAddressForCaller
             | | | | | | | | | | | | | | |--if ntdll!AvrfpAPILookupCallbacksEnabled
             | | | | | | | | | | | | | | | |--then ntdll!AVrfCallAPILookupCallback
             | | | | | | | | | | |--ntdll!LdrpInitializeNode                      /* Initialize kernel32.dll */
             | | | | | | | | | | | |--ntdll!LdrpCallInitRoutine
             | | | | | | | | | | | | |--kernel32!BaseDllInitialze                 /* Entry point kernel32.dll */
             | | | | | | | | | |--ntdll!LdrpReleaseLoaderLock
             | | | | | |--ntdll!LdrpEnableParallelLoading                          /* Load other dependencies */
             | | | | | |--ntdll!LdrpMapAndSnapDependency                           /* Map dependencies */
             | | | | | |--ntdll!LdrpDrainWorkQueue
             | | | | | |--ntdll!LdrpPrepareModuleForExecution                      /* Initialize mapped modules */
             | | | | |--ntdll!NtTestAltert                                         /* Empty APC queue */
EDR 3 hook   | |--ntdll!ZwContinue                                                /* Start execution main */
             | |--ntdll!RtlRaiseStatus
```

Figure 3: The red arrows point to the functions that EDRs hook to load their user-mode detection measures

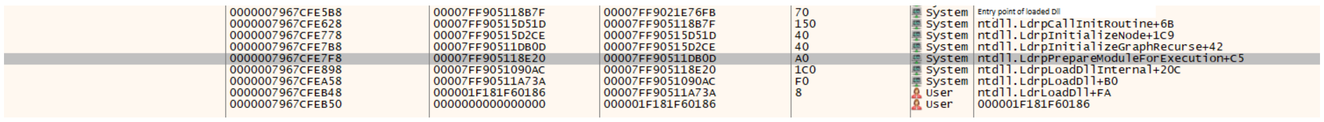
As an example, figure 4 shows the hook on `LdrLoadDll`. The initial bytes of `LdrLoadDll` are replaced with a jump instruction that points to the injected shellcode. This hooked version of `LdrLoadDll` is called as a subordinate function of `LdrInitializeThunk`. When `LdrLoadDll` executes, the flow of execution is redirected to the injected shellcode.

This shellcode is responsible for loading the EDR's detection measures. Figure 5 depicts the call stack of the shellcode that loads the EDR's hooking DLL. In the callstack, we can see that the root function is unbacked, meaning it's not part of a legitimate module, which indicates it has been injected into the process.



```
00007FFB6A5E6A10 <ntdll.LdrLoadDll>   48:B8 8601F681F1010000   mov rax,1F181F60186
00007FFB6A5E6A1A                      FFE0                     jmp rax
00007FFB6A5E6A1C                      D000                     rol byte ptr ds:[rax],1
00007FFB6A5E6A1E                      0000                     add byte ptr ds:[rax],al
00007FFB6A5E6A20                      48:8B05 E9DA1600         mov rax,qword ptr ds:[<__security_cookie>]
00007FFB6A5E6A27                      48:33C4                  xor rax,rsp
00007FFB6A5E6A2A                      48:898424 C0000000       mov qword ptr ss:[rsp+C0],rax
00007FFB6A5E6A32                      4D:8BF1                  mov r14,r9
00007FFB6A5E6A35                      49:8BF8                  mov rdi,r8
00007FFB6A5E6A38                      4C:8BD2                  mov r10,rdx
00007FFB6A5E6A3B                      48:8BF1                  mov rsi,rcx
```

Figure 4: `LdrLoadDll`'s initial bytes have been replaced with a jump instruction to the EDR's shellcode



```
0000007967CFE5B8   00007FF905118B7F   00007FF9021E76FB   70    System   Entry point of loaded Dll
0000007967CFE628   00007FF90515D51D   00007FF905118B7F   150   System   ntdll.LdrpCallInitRoutine+6B
0000007967CFE778   00007FF90515D2CE   00007FF90515D51D   40    System   ntdll.LdrpInitializeNode+1C9
0000007967CFE7B8   00007FF90511DB0D   00007FF90515D2CE   40    System   ntdll.LdrpInitializeGraphRecurse+42
0000007967CFE7F8   00007FF905118E20   00007FF90511DB0D   A0    System   ntdll.LdrpPrepareModuleForExecution+C5
0000007967CFE898   00007FF9051090AC   00007FF905118E20   1C0   System   ntdll.LdrpLoadDllInternal+20C
0000007967CFEA58   00007FF90511A73A   00007FF9051090AC   F0    System   ntdll.LdrpLoadDll+B0
0000007967CFEB48   000001F181F60186   00007FF90511A73A   8     User     ntdll.LdrLoadDll+FA
0000007967CFEB50   0000000000000000   000001F181F60186         User     000001F181F60186
```

Figure 5: Callstack of the EDR's shellcode loading it's hooking DLL

The injected shellcode writes the path and name of the hooking DLL into `rcx` and `r9` (following the x64 fastcall convention) and then invokes `LdrLoadDll`. Once the hooking DLL is loaded, its entry point (e.g. `DllMain`) is executed, which is responsible for initiating the hooks on critical functions. After `LdrLoad` completes, the shellcode removes the inline hook and resumes the process's normal execution flow. From this point, the EDR can intercept API calls in real-time and monitor the process.

In the call graph, we can observe the first time `LdrLoadDll`, `LdrpLoadDll`, and `NtContinue` execute. At one of these points, depending on the specific EDR, the EDR's detection measures (e.g., hooking DLL) are loaded.

For EDRs hooking `NtContinue`, techniques like Early Bird APC and Early Cascade Injection preempt the EDR's detection measures. This means that the malicious code (e.g. implant) runs before the detection measures are loaded. In the call graph, we can see that `NtTestAlert` is executed before `NtContinue`. Since `NtTestAlert` empties the APC queue, it ensures that the malicious code runs before the EDR's detection measures are active.

For EDRs hooking `LdrLoadDll` and `LdrpLoadDll` the EDR takes control early in the process, at the loading of `kernel32.dll`, which is before `g_pfnSE_DllLoaded`. This allows the EDR to gain control over the process before we do. However, as we can see in the call graph `g_pfnSE_DllLoaded` provides us with control before the initialization of the EDR, at that point we take control. This means that, despite the EDR taking control first, we can still disrupt the initialization of its detection measures as we can take control before the DLL initializes, preventing the EDR from loading them.

We also observed that most EDRs, through the shellcode, initially load `kernel32.dll` and `kernelbase.dll`, following the normal execution flow. Afterward, they load their hooking DLL via `LdrLoadDll`. Remember that `g_pfnSE_DllLoaded` is executed during the initialization part of `LdrLoadDLL`, in this case for `kernelbase.dll`. That is well-before the EDR's detection measures are loaded by the shellcode. In theory, at this stage, we can remove the hook on `LdrLoadDll`, revert to the original code path for loading `kernel32.dll`, and proceed with execution, bypassing the EDR's loading process.

There are likely numerous ways to prevent user-mode EDR detection measures using the callback pointers discussed in this blog. We've presented one potential approach, which could be integrated into Early Cascade Injection by leveraging the `g_pfnSE_DllLoaded` callback pointer. This would allow an implant injected via Early Cascade Injection to run more stealthily, further evading EDR detection.

Conclusion

In this blog, we explored how a process is created in Windows, focusing on the user-mode part of process creation. We presented a call graph that outlines the key events during process creation. We then examined how Early Bird APC Injection works and interacts with the user-mode part, specifically when the queued APC is executed. After that, we discussed EDR-Preloading, which showed us how we can achieve code execution during process creation simply by overwriting a pointer. This led us to further investigate and discover a new pointer. However, it wasn't possible to execute fully functional code through it. By combining the APC queuing element of Early Bird APC with the new pointer, inspired by EDR-Preloading, we developed and explained Early Cascade Injection. Finally, we highlighted the key features of this technique. I hope you found the call graph as informative as we did – providing an overview of the process creation, revealing the timing of EDR security measures, and showing how Early Cascade Injection interacts with process creation.

Implementation in Outflank Security Tooling

Due to the strong OPSEC properties of this research, we need to prevent misuse and thus will not make the source code of this project public. However, Early Cascade Injection and all other parts of this research are already available for our vetted Outflank Security Tooling (OST) community. Interested in a demo? Subscribe here.

Thank you for reading this blog post, and we hope you learned something new!

References

[1] Bypassing EDRs With EDR-Preloading – Marcus Hutchins
[2] Process Creation Flags – MSDN
[3] Windows 10 Parallel Loading Breakdown – BlackBerry
[4] New 'Early Bird' Code Injection Technique Discovered – Cyberbit
[5] Microsoft is building new Windows security features to prevent another CrowdStrike incident – The Verge
[6] NtQueueApcThread – NTAPI Undocumented Functions – NTInternals
[7] AlternativeShellcodeExec – aahmad097
[8] Critical Section Objects – MSDN
[9] What is Loader Lock? – Elliot Killick
[10] Removing Kernel Callbacks Using Signed Drivers – br-sn
[11] Demystifying Shims – or – Using the App Compat Toolkit to make your old stuff work with your new stuff – MSDN