Exception Junction - Where All Exceptions Meet Their Handler

bruteratel.com/research/2024/10/20/Exception-Junction

Chetan Nayak

October 20, 2024

This blog is in relation to some of the hurdles I've met while debugging and researching various new features for Brute Ratel. Before we get started, let me inform you that this blog is not for beginners. It requires some knowledge about Windows internals, exception handlers, and getting your hands dirty with a debugger, preferably x64dbg. And to add to that, there's limited to near zero information on the web related to this topic, thus I spent the last 24 hours researching and writing this from scratch while being high on caffeine.

There are two parts to this blog. The first part contains the 'what', 'how', and 'why' I reached here, and the second part focuses on the solution. And before we get started, I would like to thank Elastic EDR for making my life this hard :).

Part I: ‘The What?’

What are exception handlers?

There are multiple types of exception handlers, but we will be focusing on Vectored Exceptions (VEH). Both exception handlers (general) and vectored exception handlers in Windows serve the purpose of handling exceptions, such as access violations or divide-by-zero errors. However, they differ in their mechanisms, priority, and use cases. Structured Exception Handling (SEH) is the default mechanism for handling exceptions in Windows programs. An exception handler is registered within a specific function using constructs like ‘`__try / __except`’ mechanism. SEH is local to the function and cannot handle all exceptions globally unless explicitly configured via an exception filter.

A vectored exception handler (VEH) is a global exception handler mechanism introduced in Windows. Unlike SEH, vectored handlers are not stack-based but registered globally within the process using ‘`kernel32!AddVectoredExceptionHandler`’/’`ntdll!RtlAddVectoredExceptionHandler)`’. Vectored handlers are called before SEH and multiple vectored handlers can exist, unlike SEH, which is limited by stack frames.

Part II: ‘The How?’

How did I reach here?

Vectored exception handlers can be used for a variety of purposes. In my case, it was mostly related to debugging and anti-debugging. Brute Ratel is an extremely large project and one of the main tasks for the release of 2.1 was to make sure every NTAPI function call that goes to the kernel has a valid call stack. While building a call stack is easy, finding every Windows API function used by the Badger, and pivoting them via call stack spoofing was a tedious task. I rewrote one of my old code for Process Instrumentation hook back from 2021 and built a DLL which can be loaded before the start of any process, for dumping information about every NTAPI->Syscall being performed. Here is the PI Tracker code:

The NTAPI-Syscall Tracker (pi-tracker.c):

```
#include "windows.h"
#include "stdio.h"

#define EXPORT __declspec(dllexport)
HANDLE hModule;
EXPORT BOOL PIHookEnable();
```

```
EXPORT BOOL PIHookDisable();
BOOL PIHook(BOOL enable);
VOID GetSyscallName(FARPROC SyscallRet);
extern void hookedCallback();
extern NTSTATUS NtSetInformationProcess();

#ifndef NT_SUCCESS
#define NT_SUCCESS(status) ((NTSTATUS) (status) >= 0)
#endif
#define ProcessInstrumentationCallback 0x28
typedef struct _PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION
{
    ULONG Version;
    ULONG Reserved;
    PVOID Callback;
} PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION, *PPROCESS_INSTRUMENTATION_CALLBACK_INF(

EXPORT BOOL PIHookEnable() {
    return PIHook(TRUE);
}
EXPORT BOOL PIHookDisable() {
    return PIHook(FALSE);
}

BOOL PIHook(BOOL enable) {
    PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION InstrumentationCallbackInfo;
    InstrumentationCallbackInfo.Version  = 0;
    InstrumentationCallbackInfo.Reserved = 0;
    InstrumentationCallbackInfo.Callback = NULL;
    if (enable) {
        InstrumentationCallbackInfo.Callback = hookedCallback;
    }
    if (NT_SUCCESS(NtSetInformationProcess((HANDLE)-1, ProcessInstrumentationCallback,
        return TRUE;
    }
    return FALSE;
}

VOID GetSyscallName(FARPROC SyscallRet) {
    PIHook(FALSE);
    FARPROC funcPtr = SyscallRet - 0x14;
    BYTE* baseAddress = (BYTE*)hModule;
    char* functionName = NULL;
    IMAGE_DOS_HEADER* dosHeader = (IMAGE_DOS_HEADER*)baseAddress;
    if (dosHeader->e_magic != IMAGE_DOS_SIGNATURE) {
        goto cleanUp;
    }
    IMAGE_NT_HEADERS* ntHeaders = (IMAGE_NT_HEADERS*)(baseAddress + dosHeader->e_lfanew
    if (ntHeaders->Signature != IMAGE_NT_SIGNATURE) {
```

```
        goto cleanUp;
    }
    IMAGE_OPTIONAL_HEADER* optionalHeader = &ntHeaders->OptionalHeader;
    IMAGE_DATA_DIRECTORY* exportDataDir = &optionalHeader->DataDirectory[IMAGE_DIRECTOR
    if (exportDataDir->VirtualAddress == 0) {
        goto cleanUp;
    }
    IMAGE_EXPORT_DIRECTORY* exportDirectory = (IMAGE_EXPORT_DIRECTORY*)(baseAddress + e
    DWORD* funcAddressArray = (DWORD*)(baseAddress + exportDirectory->AddressOfFunctio
    DWORD* nameArray = (DWORD*)(baseAddress + exportDirectory->AddressOfNames);
    WORD* ordinalArray = (WORD*)(baseAddress + exportDirectory->AddressOfNameOrdinals)
    for (DWORD i = 0; i < exportDirectory->NumberOfFunctions; i++) {
        FARPROC currentFunction = (FARPROC)(baseAddress + funcAddressArray[i]);
        if (currentFunction == funcPtr) {
            for (DWORD j = 0; j < exportDirectory->NumberOfNames; j++) {
                if (ordinalArray[j] == i) {
                    functionName = (char*)(baseAddress + nameArray[j]);
                    goto cleanUp;
                }
            }
        }
    }
cleanUp:
    if (functionName) {
        printf("[PI-TRACKER] %s (%p)\n", functionName, funcPtr);
    }
    PIHook(TRUE);
}

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD dwReason, LPVOID lpReserved)
{
    switch (dwReason){
    case DLL_PROCESS_ATTACH: {
        hModule = GetModuleHandleA("ntdll");
        PIHookEnable();
        break;
    }
    case DLL_PROCESS_DETACH:
        PIHookDisable();
        break;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
        break;
    }
    return TRUE;
}
```

The Hook (hook.asm):

```
section .text
global hookedCallback:
extern GetSyscallName

hookedCallback:
    push rcx
    push r10
    mov rcx, r10
    call GetSyscallName
    pop r10
    pop rcx
    jmp r10
```

The MakeFile:

```
make:
        nasm -f win64 hook.asm -o hook.o
        x86_64-w64-mingw32-gcc pi_tracker.c hook.o -o PI-Tracker.dll -s -O2 -lntdll -l
```

I won't be deep diving into the above code, because it's kinda straightforward. A brief overview would be, that the PI-Tracker.c file, has a Dllmain function that enables '`NTAPI->Syscall`' tracking whenever it is loaded. It calls '`NtSetInformationProcess`' API call with a callback hook and '`ProcessInstrumentationCallback`' class. Once this is executed, every time an NTAPI->Syscall is called, before returning from the kernel to the return address of the syscall, the kernel makes a jump to the userland callback hook ('`hookedCallback`'). One thing I noticed when this hook was executed, was that the '`R10`' register contains the original return address of the syscall. And since every syscall return address (in Windows 10) is 0x14 bytes away from the actual NTAPI instruction, I can just subtract and find the NTAPI pointer. Once I have this, I can walk through the Export Address Table (EAT) of the '`ntdll.dll`' to find which API was called by doing an ordinal comparison.

Once this DLL is compiled, it can be loaded into any process using '`LoadLibraryA("pi-tracker.dll")`' API call and it will dump all the NTAPI->Syscalls being called. Brute Ratel uses Vectored handlers for a variety of tasks from anti-debugging to walking some EDR's DLL and finding the original overwritten syscall values etc. You can enable Vectored Exception using '`kernel32!AddVectoredExceptionHandler`' which calls '`ntdll!RtlAddVectoredExceptionHandler`'. While tracking this API call, I found that '`ntdll!RtlAddVectoredExceptionHandler`' calls '`ntdll!NtProtectVirtualMemory`'. Now this was kind of concerning to me because Brute Ratel spoofs the stack of every '`ntdll!NtProtectVirtualMemory`', but not the first execution of '`ntdll!RtlAddVectoredExceptionHandler`' as I use this API for some parts of call stack spoofing. So the problem is that the first execution of '`ntdll!RtlAddVectoredExceptionHandler`' will run with an unbacked stack which would call

‘ntdll!NtProtectVirtualMemory’ creating a problem for an EDR like Elastic (with stack rules loaded) which relies heavily on call stack analysis. I come from a background where I spent my initial days reversing on Windows 7, and this part of calling ‘ntdll!NtProtectVirtualMemory’ was not present in Windows 7. Being curious, I decided to see ‘WHY’ ‘ntdll!RtlAddVectoredExceptionHandler’ calls ‘ntdll!NtProtectVirtualMemory’ and if that can be avoided as this isn’t present on the previous versions of Windows. So if I could rewrite the entire ‘ntdll!RtlAddVectoredExceptionHandler’ from scratch and avoiding the call of ‘ntdll!NtProtectVirtualMemory’ like in Windows 7, then my problem is solved. And for those of you folks, who would plan to visit unknowncheats.me or reactos for this, let me inform you the entire code for that is buggy and does not support anything post the earlier versions of Windows 7.

Part III: ‘The Why?’

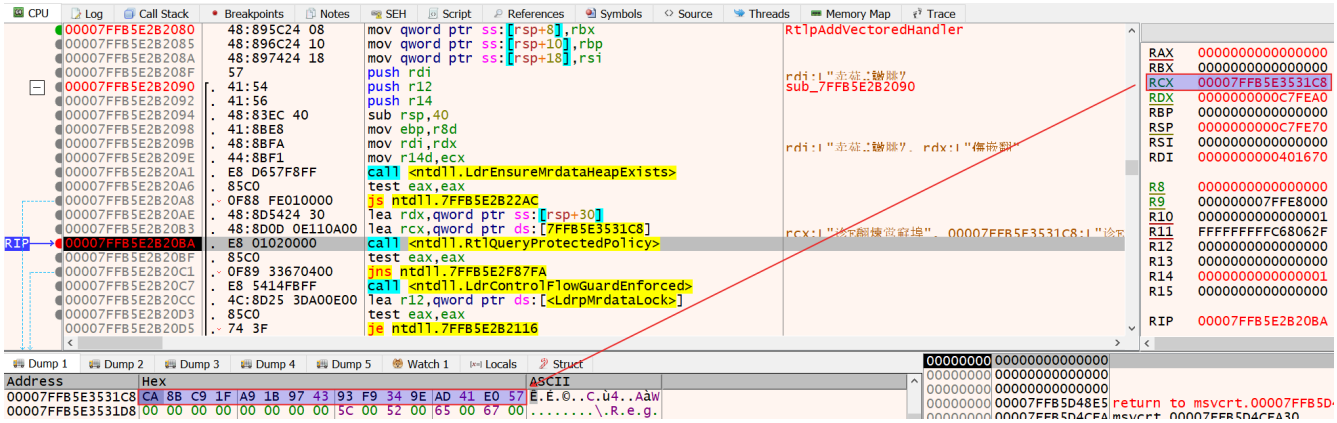
Why did I reach here?

So to understand why the ‘ntdll!NtProtectVirtualMemory’ is being called, I decided to reverse the entirety of the ‘ntdll!RtlAddVectoredExceptionHandler’ API call. There are a lot of low-level structure modifications that I just built on the fly through my logic, and they could be wrong. But it works across various versions of Windows, so don’t fix what’s not broken I guess. I decided to download the symbols for ntdll in x64dbg so that I understand the internals properly. Note that any function that starts with ‘Ldrp/Rtlp’ is an internal function of the ntdll and does not have an export. They can only be viewed by downloading the PDB symbols, to call them, you need to perform a pattern-based search which could change over different versions of ‘ntdll.dll’ (more on this later at the end of the blog).

The below figure shows that calling ‘ntdll!RtlAddVectoredExceptionHandler’ calls ‘ntdll!RtlpAddVectoredHandler’ (notice the jump). There are two arguments to ‘ntdll!RtlAddVectoredExceptionHandler’ which reside in RCX and RDX register. The first argument (RCX) ideally contains a ULONG value which specifies whether the handler should be added to the start of a LinkedList or the end. A non-zero value will add the new handler to the start and vice versa. In the below image, I am passing ‘1’ to RCX and my function pointer as a VectoredHandler to ‘RDX’. My VectoredHandler is built to simply catch the exception, print that caught it, and exit.

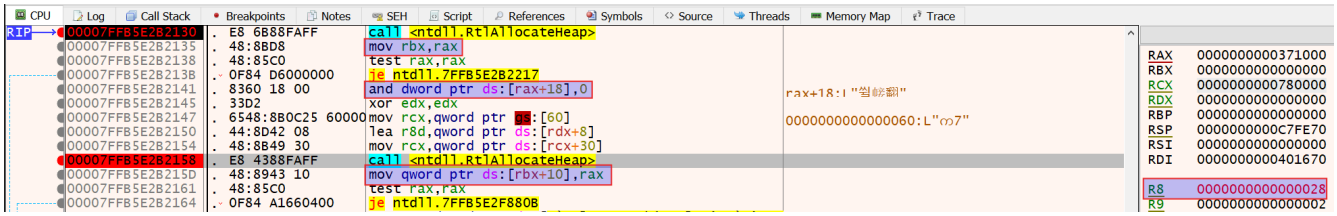
The ‘ntdll!RtlpAddVectoredHandler’ API calls ‘ntdll!LdrEnsureMrdataHeapExists’ to check if a heap was already created on a previous iteration of ‘ntdll!RtlAddVectoredExceptionHandler’ call. If this is the first call to ‘ntdll!RtlAddVectoredExceptionHandler’, then ‘ntdll!LdrEnsureMrdataHeapExists’ returns False. ‘ntdll!LdrEnsureMrdataHeapExists’ also stores a LinkedList (‘ntdll!LdrpVectorHandlerList’) built on heap. If this LinkedList does not exist, it returns False, else True. ‘ntdll!LdrEnsureMrdataHeapExists’ also calls ‘ntdll!LdrControlFlowGuardEnforced’ to make sure this is not being exploited by some rop-gadget and is a legitimate call. When ‘ntdll!LdrControlFlowGuardEnforced’ returns TRUE, it indicates that CFG is enforced, meaning that the system verifies indirect call targets to ensure they are legitimate before the call happens. This function typically checks the ‘ntdll!LdrSystemDllInitBlock’, which contains CFG-related flags and bitmap data to determine whether CFG is active for the current process or module. ‘ntdll!LdrControlFlowGuardEnforced’ is called multiple times across this code to ensure rop-gadgets are not exploited.

After checking the CFG, it calls ‘ntdll!RtlQueryProtectedPolicy’ with GUID ‘{ 0x1FC98BCA, 0x1BA9, 0x4397, {0x93, 0xF9, 0x34, 0x9E, 0xAD, 0x41, 0xE0, 0x57} }’ to query if VEH is enabled. If not, it returns STATUS_NOT_FOUND. The below image shows the call to ‘ntdll!RtlQueryProtectedPolicy’ with the first argument (RCX) as the pointer to the GUID ‘CA 8B C9 1F A9 1B 97 43 93 F9 34 9E AD 41 E0 57’ in the dump section below.



The ‘ntdll!LdrControlFlowGuardEnforced’ is called again to perform CFG validation.

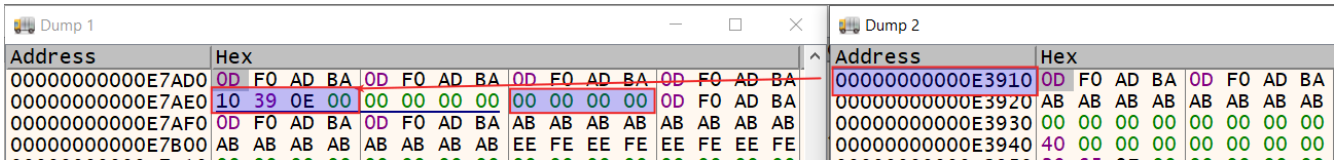
Next, it calls '`ntdll!RtlAllocateHeap`' twice. The first call allocates Heap (we will call it FirstHeap - Address:0xE7AD0) of 0x28 (40) bytes. The second Allocation (SecondHeap - Address:0xE3910) is of 8 bytes. If you see the image below, once the FirstHeap (0xE7AD0) is allocated, the FirstHeap (0xE7AD0) pointer is moved to 'RBX' from 'RAX' and then the 0x18th offset of this heap is filled with zeroes using the 'and dword' operation (4 bytes zeroed out).



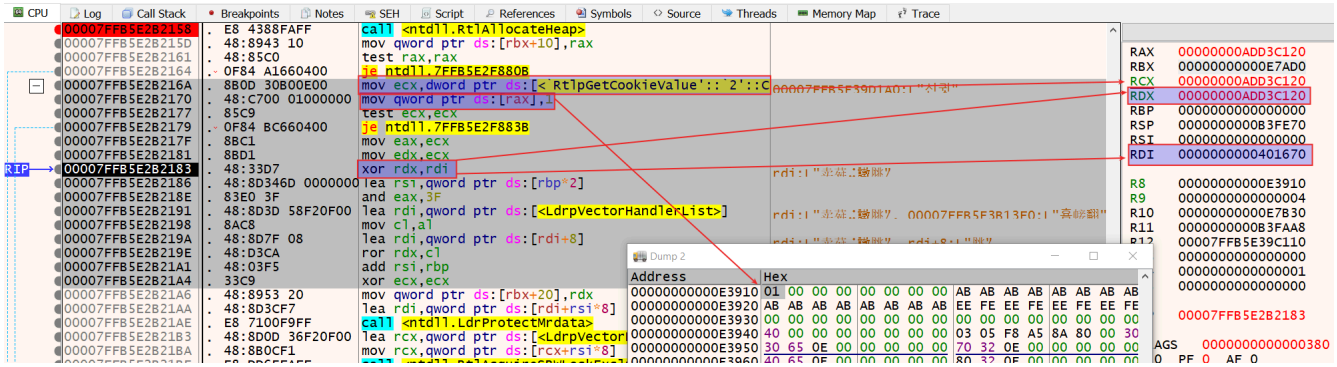
I couldn't find any correct structure information on the web for this heap, so after a lot of trial and error, this is the structure of the FirstHeap (40/0x28 bytes) I created. This is also the heap that is returned by '`ntdll!RtlAddVectoredExceptionHandler`' at the end of the function, and thus we will call it '`VECTORED_HANDLER_ENTRY`' structure. This structure contains LinkedLists as can be seen below. Also, note that the '`PVOID Refs`' is nothing but the pointer to the SecondHeap (0xE3910) which we allocated after the FirstHeap allocation (0xE7AD0).

```
typedef struct _VECTORED_HANDLER_ENTRY {
    struct _VECTORED_HANDLER_ENTRY *pNext;
    struct _VECTORED_HANDLER_ENTRY *pPrev;
    PVOID Refs;
    ULONG Padding0;
    PVECTORED_EXCEPTION_HANDLER pVectoredHandler;
} VECTORED_HANDLER_ENTRY, *PVECTORED_HANDLER_ENTRY;
```

As can be seen below, the SecondHeap pointer (0xE3910) is moved to the 0x10th offset(PVOID Refs) of the FirstHeap (0xE7AD0).



The '`<ntdll!RtlpGetCookieValue>`' stores a unique 4-byte Cookie (0xADD3C120). This is a random DWORD value when a process is created and is never the same. This value is used by the inline function '`ntdll!RtlEncodePointer`' to encode the VectoredHandler pointer. You will not see the '`ntdll!RtlEncodePointer`' function being called itself below, because it's running inline. You can also see the value 'one' being moved to the SecondHeap (PVOID Refs -> 0xE3910).



The ‘xor rdx, rdi’ instruction xor’s the VectoredHandler’s pointer (0x401670) with DWORD cookie (0xADD3C120). Once xor’d, it rotates the xor’d value for an ‘x’ number of times. This ‘x’ time is the last byte value of the cookie itself (0x20). This is the entire ‘ntdll!RtlEncodePointer’ process. Once this ROR operation is complete, it is moved to the last 8 bytes of the FirstHeap (0xE7AD0).

Next, ‘ntdll!LdrProtectMrData’ is called to change the protection of ‘.mrdata’ section in ntdll.dll to ReadWrite. This is by default ReadOnly. ‘ntdll!LdrProtectMrData’ uses ‘ntdll!NtProtectVirtualMemory’ to change the permission from ReadOnly to ReadWrite. This is the part that I was trying to figure out. Apparently, in the older versions of Windows, this section didn’t exist. However, from Windows 8.1, the ‘.mrdata’ section was added to ntdll. This section stores mutable runtime data that needs to be protected during normal execution but occasionally modified. It holds structures like the ‘ntdll!LdrSystemDllInitBlock’, which contains data critical for managing system mitigations like Control Flow Guard (CFG). This section begins as writable but is later set to read-only to prevent tampering, with temporary unprotection allowed when needed during execution. This section also stores exception handler data (‘ntdll!LdrpVectorHandlerList’ struct) in a ReadOnly state, but we don’t know the offset for this yet, only Windows does. Since we are adding a new VectoredHandler, the pointer for this handler needs to be added to the VECTORED_HANDLER_ENTRY struct which we created above (FirstHeap (0xE7AD0)), and this data is then written to the ‘.mrdata->ntdll!LdrpVectorHandlerList’ section. This means I do not have any option to skip the usage of ‘ntdll!NtProtectVirtualMemory’ which I initially thought I did from my experience with Windows 7.

The ‘ntdll!LdrProtectMrData’ first moves ‘ntdll!LdrpMrdataLock’ to RCX and calls ‘ntdll!RtlAcquireSRWLockExclusive’ to lock the section before changing permission. Next, it calls ‘ntdll!LdrpChangeMrdataProtection’ which calls ‘ntdll!NtProtectVirtualMemory’ to change the permission of ‘.mrdata’ from ReadOnly to ReadWrite. The ‘ntdll!LdrProtectMrData’ then calls ‘ntdll!RtlReleaseSRWLockExclusive’ to release ‘ntdll!LdrpMrdataLock’. Once ReadWrite is enabled, we need to find the address for ‘ntdll!LdrpVectorHandlerList’ (NOTE: THERES A CATCH). Ideally Windows knows the

address for this which is different in different versions of ntdll. So, Windows can easily extract this information, but if we have to do it manually, then we will need to perform pattern-based hunting (More on this later). For now, Once this structure is extracted, The first pointer present in the '`ntdll!LdrpVectorHandlerList`' is moved to RCX which is the '`ntdll!LdrpVehLock`'. Windows needs to perform a lock on this using '`ntdll!RtlAcquireSRWLockExclusive(LdrpVehLock)`' without which writing to this section can cause race condition crashes due if reading and writing occurs at the same time. Once the writing is complete, '`ntdll!RtlReleaseSRWLockExclusive`' is called to release this lock.

Once this list is extracted, a check is performed. If the second pointer in the '`ntdll!LdrpVectorHandlerList`' is the same as self, then a PEB flag (CrossProcessFlags [PEB+0x50]) is enabled for VEH. This is done by calling the '`InterlockedBitTestAndSet`' API call inline, which sets the flag value to 2.

Another check is performed on the first parameter passed to '`ntdll!RtlAddVectoredExceptionHandler`'. If this value is non-zero, then the VectoredHandler needs to be called as the first handler, else last. To add a new handler to the start of the '`ntdll!LdrpVectorHandlerList`', we need to configure the ListEntries. A quick code would look like this where LdrpVectorHandlerList is the original list, and pNewVehEntry is our FirstHeap (0xE7AD0) buffer:

```
if(FirstHandler) { //Add new node to the head of VEH
    pNewVehEntry->pNext = LdrpVectorHandlerList->pFirstHandler;
    pNewVehEntry->pPrev = (PVECTORED_HANDLER_ENTRY)&LdrpVectorHandlerList->pFirstHandle
    LdrpVectorHandlerList->pFirstHandler->pPrev = pNewVehEntry;
    LdrpVectorHandlerList->pFirstHandler = pNewVehEntry;
} else { //Add new node to the end of VEH
    pNewVehEntry->pNext = (PVECTORED_HANDLER_ENTRY)&LdrpVectorHandlerList->pFirstHandle
    pNewVehEntry->pPrev = LdrpVectorHandlerList->pLastHandler;
    LdrpVectorHandlerList->pLastHandler->pNext = pNewVehEntry;
    LdrpVectorHandlerList->pLastHandler = pNewVehEntry;
}
```

In brief, if the argument provided to the '`ntdll!RtlAddVectoredExceptionHandler`' is non-zero, then:

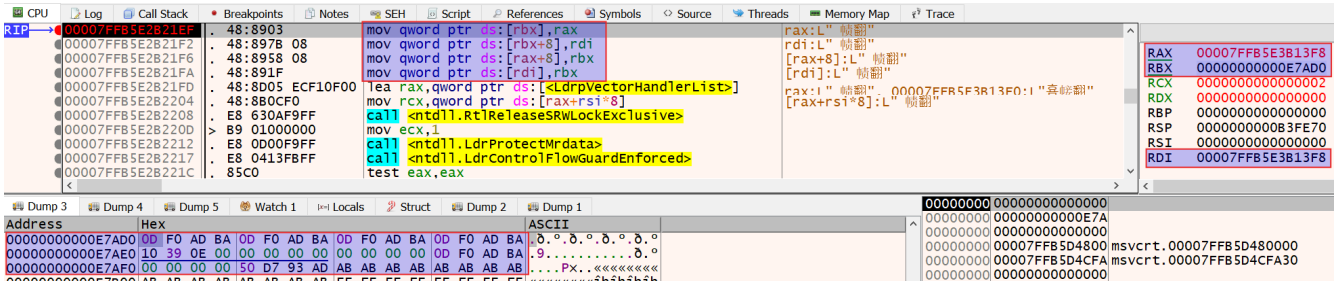
1. A new node (pNewVehEntry) is added at the beginning of the list. This '`pNewVehEntry`' is the same FirstHeap (0xE7AD0) we allocated above.
2. It points to the current first handler, and the current first handler's ListEntry (pPrev) is updated to point back to the new node.
3. The list's 'pFirstHandler' is updated to the new node.

If the argument provided is zero, then:

1. The new node (pNewVehEntry) is added at the end of the list.
2. It points to the start (‘pFirstHandler’ pointer) and the current last handler.
3. The last handler’s ‘pNext’ is updated to the new node, and ‘pLastHandler’ is set to the new node.

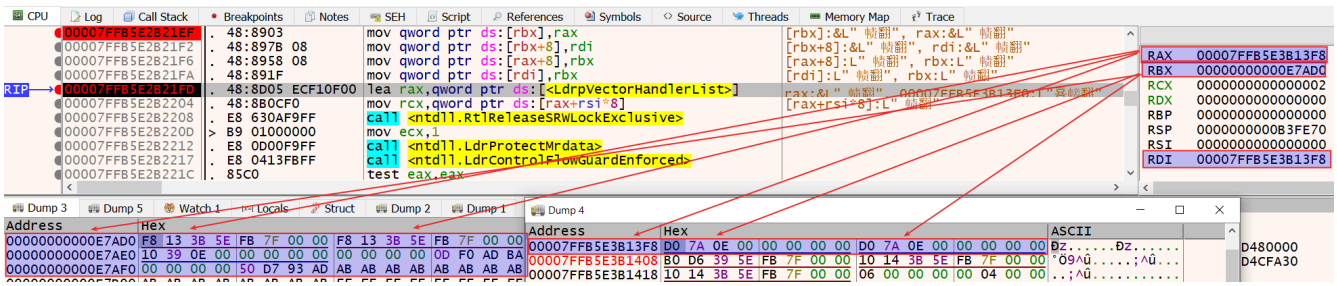
In our case, we supplied a non-zero value, so our handler should get added to the start of the ‘`ntdll!LdrpVectorHandlerList`’. Here is a picture of before and after the list shuffling. You can check the FirstHeap (0xE7AD0) in RBX and bottom left dump, and original handler list in RAX and bottom right dump.

Before Shuffling:



1. Original first handler is 0x7FFC312D13F8 (RAX) which was initially extracted from the ‘`ntdll!LdrpVectorHandlerList`’ (second offset of 8 bytes - this was garbage probably during initialization)
2. Second handler is 0x7FFC312D13F8 (RDI). Both handlers are the same because there is no handler enabled yet.
3. Original first handler list is moved to the start of the FirstHeap (0xE7AD0) [rbx]
4. Second handler list is moved to the second offset of 8 bytes in the FirstHeap (0xE7AD0) list [rbx+8]
5. FirstHeap (0xE7AD0) is moved to the second offset of 8 bytes in the first handler list [rax+8]
6. FirstHeap (0xE7AD0) is moved to the first offset of 8 bytes in the first handler list [rax+8]

After Shuffling:



In short, our FirstHeap (0xE7AD0) structure now looks like this:

```
typedef struct _VECTORED_HANDLER_ENTRY {
    struct _VECTORED_HANDLER_ENTRY *pNext;          // 0x7FFB5E3B13F8 - First Handler I
    struct _VECTORED_HANDLER_ENTRY *pPrev;          // 0x7FFB5E3B13F8 - Next Handler L:
    PVOID Refs;                                     // 0x0E3910 - Pointer to SecondHeaı
    ULONG Padding0;                                 // Garbage - No idea what this is u
    PVECTORED_EXCEPTION_HANDLER pVectoredHandler;   // Encoded Pointer for VectoredHanc
} VECTORED_HANDLER_ENTRY, *PVECTORED_HANDLER_ENTRY;
```

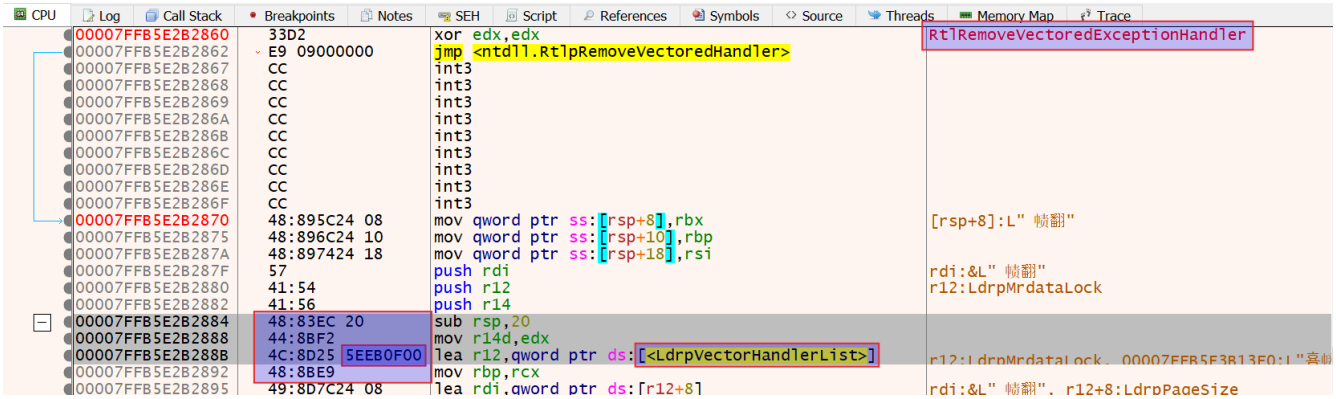
This dump on the top left in the image is the final buffer that gets returned by '`ntdll!RtlAddVectoredExceptionHandler`' once it completes. Once all the writing is complete, '`ntdll!RtlReleaseSRWLockExclusive`' is called with '`ntdll!LdrpVehLock`' to release the VEH Lock. '`ntdll!LdrProtectMrdata`' is called again to reset the ReadOnly permission from ReadWrite for '.mrdata', and a final '`ntdll!LdrControlFlowGuardEnforced`' is called again to perform CFG validation, before returning the FirstHeap (0xE7AD0) to the user.

Part IV: Solution?

Unhandled Exception

So what was the solution to my problem? There was no solution. This is just another day as a Brute Ratel developer where I spend a ton of time reversing something, which may or may not be of much consequence. However, I did write an entire custom AddVEHHandler named '[RtlpAddVectoredExceptionHandler]' which is the first custom handler ever written. And yes, this isn't available on 'unknowncheats.me' or 'reactos' source code (atleast I couldn't find it on web, or I am bad at google. lol). That was the first place I looked, only to find despair in return.

However, there is one interesting part in the above code. In case of the original '`ntdll!RtlAddVectoredExceptionHandler`', Windows knows the address of the original first handler ('`ntdll!LdrpVectorHandlerList`') from which it extracts the '`ntdll!LdrpVehLock`' and then updates it. If we are writing our own VEH handler code, then we will need to find this list. After a bit more digging, I found that '`ntdll!RtlRemoveVectoredExceptionHandler`' stores this information and is easy to find using a pattern of '48 83 EC 20 44 8B ?? ?? 8D ?? ?? ?? ?? ?? ?? 48 8B E9'.



So, we can simply search the above pattern in the text section, extract the offset dynamically and add it to the current instruction pointer to get the address of '`ntdll!LdrpVectorHandlerList`'. The below code can be used to find the pattern and is a part of the above project.

```
PVOID GetLdrpVectorHandlerList() {
    // Byte pattern for LdrpVectorHandlerList for windows 10 is: 48 83 EC 20 44 8B ? ?
    // Pattern to search for: 0x4883EC20448BF24C8D254EEB0F00 (last 4 bytes are the offs
    const BYTE pattern[] = { 0x48, 0x83, 0xEC, 0x20, 0x44, 0x8B, 0xF2, 0x4C, 0x8D, 0x25
    const size_t patternLength = sizeof(pattern);
    UINT_PTR hNtdll = findNtdll();

    PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)hNtdll;
    PIMAGE_NT_HEADERS ntHeader = (PIMAGE_NT_HEADERS)((BYTE*)hNtdll + dosHeader->e_lfane
    PIMAGE_SECTION_HEADER textSection = IMAGE_FIRST_SECTION(ntHeader);
    for (int i = 0; i < ntHeader->FileHeader.NumberOfSections; i++) {
        if (strncmp((const char*)textSection->Name, ".text", 5) == 0) {
            break;
        }
        textSection++;
    }
    BYTE* textSectionStart = (BYTE*)hNtdll + textSection->VirtualAddress;
    DWORD textSectionSize = textSection->Misc.VirtualSize;
    for (DWORD i = 0; i < textSectionSize - patternLength; i++) {
        if (memcmp(textSectionStart + i, pattern, patternLength) == 0) {
            int32_t offset = *(int32_t*)(textSectionStart + i + patternLength);
            BYTE* instruction_after_offset = textSectionStart + i + patternLength + 4;
            BYTE* ldrpVehList = instruction_after_offset + offset;
            return ldrpVehList;
        }
    }
    return NULL;
}
```

I've tested this against different ntdll versions and it worked, but it doesn't work on Windows 7 as the pattern is different for that. It needs a different offset too, which I am sure you will find

out if you are still reading this blog. So, that concludes this blog and an intro to the life of a ‘Real C2 Developer’ (pardon my shade XD).

Social Responsibility to Provide Detection Rule

For detection engineers, you can check ‘`ntdll!NtProtectVirtualMemory`’ call and the address of the ‘.mrdata’ section as a parameter which can be a big anomaly in itself, but for the ones who love yara rules, heres a thought.

```
rule HexBytePatternMatch
{
    meta:
        description = "Detect pattern in anything which is not ntdll"
        author = "Paranoid Ninja"
        date = "2024-10-12"

    strings:
        $pattern = { 48 83 EC 20 44 8B ?? ?? 8D ?? ?? ?? ?? ?? 48 8B E9 }

    condition:
        $pattern
}
```

There are a few other ways on how one write to ‘.mrdata’ without being detected, but thats for some other day. Hope you guys enjoyed the blog, and theres much more in the arsenal of BRc4 2.1 Stay tuned and happy hacking :).