MutationGate

Background

Motivation

Considering inline hook is a major detection utilized by EDR products, bypassing them is an interesting topic to me. Regarding bypassing inline hook placed by EDR, there are already quite a few approaches available. Although some of the earlier approaches have decent detection rules, there are also a few mature approaches in existence. Nonetheless, I think it would be very fun to discover a new approach to bypass hook, hopefully, it can bring some improvements or advantages.

Inline Hook

EDR products like placing inline hooks at NTAPIs that are usually leveraged in malware, like **NtAllocateVirtualMemory**, **NtUnmapViewOfSection**, **NtWriteVirtualMemory**, etc. Because an NTAPI is a bridge between user space and kernel space.

image.png
Image not found or type unknown

For instance, **NtAllocateVirtualMemory** is the NTAPI version of **VirtualAlloc**. By placing an unconditional jump instruction at the NTAPI, no matter whether the program calls Win32 API or

NTAPI, the EDR is able to inspect the call and infer the intention further.

image.png Image not found or type unknown

The following screenshots show what a hooked NTAPI looks like:

image.png Image not found or type unknown

image.png Image not found or type unknown

image.png Image not found or type unknown

image.png Image not found or type unknown

And if an NTAPI is not hooked, we can notice a very consistent pattern, which is how a syscall stub looks like:

image.png Image not found or type unknown

While EDR has multiple layers of detection, inline hook is a major one.

Hardware Breakpoint

In my article Bypass Amsi On Windows 11(https://medium.com/@gustavshen/bypass-amsi-on-windows-11-75d231b2cac6), I discovered that as long as **R8** is **0** when calling **AmsiScanBuffer**, AMSI can be bypassed. However, I said it was not easy to utilize this bypass without WinDBG.

image.png Image not found or type unknown

Actually, it is possible, by utilizing hardware breakpoint. This article(https://ethicalchaos.dev/2022/04/17/in-process-patchless-amsi-bypass/) explains how to use hardware breakpoint to achieve patchless AMSI bypass. It works by setting RAX as 0, modifying AMSI scan result argument, and setting RIP as the return address when the execution is transferred to the AmsiScanBuffer function. So in our case, we just need to set R8 as 0.

I will not cover much background knowledge about hardware breakpoint and VEH here, as you can refer to the article, and the general idea is more important.

Some of the Existing Approaches

The following approaches can be used to bypass inline hook, however, each also has its IOCs. Considering there are already a lot of articles talking about them in detail, so let's briefly go through some of the approaches that can bypass inline hook.

Direct Syscall

Each Nt version of Win32 API, such as NtAllocateVirtualMemory only requires 4 instructions to work, the set of instructions is called syscall stub. The only difference between various NTAPI is the value of their syscall number.

```
mov r10, rcx
mov rax, [SSN]
syscall
ret
```

image.png Image not found or type unknown

image.png Image not found or type unknown

In a .asm file, define the syscall stub for NtAllocateVirtualMemory

```
.code

<...Other Stubs...>

NtAllocateVirtualMemory PROC
    mov r10, rcx
    mov eax, 18h
    syscall
    ret
NtAllocateVirtualMemory ENDP

<...Other Stubs...>

end
```

Inside a header or c/cpp file, use **EXTERN_C** macro to link the function definition with the syscall stub assembly code, the name should be consistent.

```
EXTERN_C NTSTATUS NtAllocateVirtualMemory(
IN HANDLE ProcessHandle,
IN OUT PVOID * BaseAddress,
IN ULONG ZeroBits,
IN OUT PSIZE_T RegionSize,
IN ULONG AllocationType,
IN ULONG Protect);
```

In this way, we can directly initiate the syscall by calling the define function. However, this approach has suspicious IOCs:

Hardcoded syscall stub could be detected, the pattern is **4c8bd1 c7c0<DWORD> 0f05c3**. A possible Yara rule to detect direct syscall is as follows:

```
rule direct_syscall
{
    meta:
        description = "Hunt for direct syscall"

    strings:
        $s1 = {4c 8b d1 48 c7 c0 ?? ?? ?? ?? 0f 05 c3}
        $s2 = {4C 8b d1 b8 ?? ?? ?? ?? 0F 05 C3}
    condition:
        #s1 >=1 or #s2 >=1
}
```

image.png Image not found or type unknown

Though it is trivial to bypass the pattern by inserting some NOP-like instructions.

This approach has a downside, we hardcode the syscall number in the source code. It does not work well when the target organization has multiple versions of operating system, because SSNs vary on different OS versions.

Syswhisper suite resolves this issue: Syswhisper 1 detects the host's OS version and selects the correct SSN. Syswhisper 2 dynamically retrieves the SSN in run-time. Anyway, direct syscall is used by these approaches.

Without custom modification to Syswhisper2's syscall stub, a possible Yara rule to detect is as follows:

```
rule syswhisper2
{
    meta:
        description = "Hunt for syswhisper2 generated asm code"

    strings:
        $s1 = {58 48 89 4C 24 08 48 89 54 24 10 4C 89 44 24 18 4C 89 4C 24 20 48 83 EC 28 8B 0D ?? ?? 00 00
E8 ?? ?? ?? ?? 48 83 C4 28 48 8B 4C 24 08 48 8B 54 24 10 4C 8B 44 24 18 4C 8B 4C 24 20 4C 8B D1 0F 05 C3}
    condition:
        #s1 >=1
}
```

image.png Image not found or type unknown

Apart from the byte sequences pattern of a syscall stub, executing syscall instruction is abnormal in a legitimate program, i.e. the syscall should be initiated inside ntdll.dll's memory region. For instance, when calling Win32 API **SleepEx** in a C program, we can notice the call stack as follows: **sleep!main -> kernelbase!SleepEx -> ntdll!NtDelayExecution ->ntoskrnl!KeDelayExecutionThread**.

image.png Image not found or type unknown

However, if we initiate the syscall directly, the call stack is as follows: **sleep!main -> sleep!NtDelayExecution ->ntoskrnl!KeDelayExecutionThread**.

image.png Image not found or type unknown

It is easy to find out that the syscall instruction is initiated in the program instead of the ntdll module.

image.png Image not found or type unknown

Indirect Syscall

We discussed the downside of direct syscall, so we want to avoid executing syscall directly, indirect syscall is an improvement. The syscall stub's pattern is very similar to the one in direct syscall, however, instead of directly executing the syscall instruction, indirect syscall stub uses an **unconditional jump** to transfer the execution to the address of a syscall instruction.

```
NtAllocateVirtualMemory PROC
    mov r10, rcx
    mov eax, (ssn of NtAllocateVirtualMemory)
    jmp (address of a syscall instruction)
    ret
NtAllocateVirtualMemory ENDP
```

Of course, we need to fetch a valid address for a syscall instruction. Assuming an NTAPI is unhooked, we can get the syscall number at the offset of **0x4** of the function, and the instruction of syscall is at **0x12**.

image.png Image not found or type unknown

However, this brings us to the "chicken or the egg" problem. If an NTAPI is hooked, then its syscall stub will not match the syscall stub pattern, and naturally, we cannot successfully extract the SSN and the address of the syscall instruction. Fortunately, considering that syscall is a special type of call instruction that does not directly specify the address to jump to but rather determines the transferred address in kernel space based on the SSN, all we need to do is specify the correct SSN and corresponding function arguments.

Therefore, we do not have to get the address of the syscall instruction in NtAllocateVirtualMemory function. We can select an unhooked NTAPI, the one that is not typically leveraged in malware, such as NtDrawText.

image.png Image not found or type unknown

Though indirect syscall improved evasion, security products could still detect them based on some IOCs:

If using Syswhisper3 without custom modification, though fewer hardcoded bytes of the syscall stub, it is still possible to find a byte sequence pattern: **4c8bd1 41ff<DWORD> c3**

A possible Yara rule to detect is as follows:

```
rule syswhisper3
{
    meta:
        description = "Hunt for syswhispe3 generated asm code"

    strings:
        $s1 = {48 89 4c 24 08 48 89 54 24 10 4c 89 44 24 18 4c 89 4c 24 20 48 83 ec 28 b9 ?? ?? ?? ?? e8}
        $s2 = {48 83 c4 28 48 8b 4c 24 08 48 8b 54 24 10 4c 8b 44 24 18 4c 8b 4c 24 20 4c 8b d1}
    condition:
        #s1 >=1 or #s2 >=1
}
```

image.png Image not found or type unknown

Besides, though the call stack looks more legitimate, a detection rule can rely on the fact that the return address resides in **NtDrawText** function, while the executed syscall is **ntoskrnl!NtAllocateVirtualMemory**.

Overwrite the text section of loaded NTDLL

EDR hooks some NTAPI by overwriting the codes, the changes happen in the **.text** section of the ntdll module. Therefore, to recover hooked functions, we can overwrite the .text section of loaded ntdll.

To achieve it, there are multiple steps:

1. Read a fresh copy of ntdll. We can read it from disk, over the Internet, from KnownDLL directory, etc.
2. Change the page permission from **RX** to **RWX**, as the .text section is not writable by default, though we can also use WriteProcessMemory or its NTAPI to overwrite hooked code.
3. Copy the .text section from the fresh copy to the loaded module's
4. Revert the page permission.

However, there are some IOCs regarding this approach:

1. EDR could find that the hook is tamped by checking the integrity of the loaded NTDLL module.
2. We use hooked functions to perform the above actions, which could trigger alerts.
3. A memory region that has RWX permission is a serious red flag.

Overwrite hooked functions

Instead of overwriting the whole .text section, we can choose to overwrite needed functions, just like patching AmsiScanBuffer. While less modification to the loaded ntdll module, IOCs of overwriting the text section of NTDLL still apply to this approach.

There are still some other approaches to bypass inline hook, but I will not go through all of them, this article(https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/) did a great job in explaining common approaches.

MutationGate

MutationGate is a variant of indirect syscall, it is a new approach to bypass EDR's inline hook by utilizing hardware breakpoint to redirect the syscall. MutationGate works by calling an unhooked NTAPI and replacing the unhooked NTAPI's SSN with hooked NTAPI's. In this way, the syscall is redirected to the hooked NTAPI's, and the inline hook can be bypassed without loading the 2nd ntdll module or modifying bytes within the loaded ntdll's memory space. The GitHub repository is https://github.com/senzee1984/MutationGate

As we discussed before, EDR tends to set inline hooks for various NTAPI, especially those that are usually leveraged in malware, such as NtAllocVirtualMemory. While other NTAPI that are not usually leveraged in malware tend not to have inline hooks, such as NtDrawText. It is very unlikely that an EDR set inline hook for all NTAPI.

Assume NTAPI NtDrawText is not hooked, while NTAPI NtQueryInformationProcess is hooked, the steps are as follows:

1. Get the address of **NtDrawText**. It can be achieved by utilizing **GetModuleHandle** and **GetProcAddress** combination, or a custom implementation of them via PEB walking.

```
pNTDT = GetFuncByHash(ntdll, 0xA1920265);	//NtDrawText hash
pNTDTOffset_8 = (PVOID)((BYTE*)pNTDT + 0x8);	//Offset 0x8 from NtDrawText
```

2. Prepare arguments for NtQueryInformationProcess
3. Set a hardware breakpoint at **NtDrawText+0x8**, when the execution reaches this address, the SSN of NtDrawText is saved in RAX, but the syscall is not initiated yet.

```
0:000> u 0x00007FFBAD00EB68-8
ntdll!NtDrawText:
00007ffb`ad00eb60 4c8bd1          mov     r10,rcx
00007ffb`ad00eb63 b8dd000000      mov     eax,0DDh
00007ffb`ad00eb68 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffb`ad00eb70 7503            jne     ntdll!NtDrawText+0x15 (00007ffb`ad00eb75)
00007ffb`ad00eb72 0f05            syscall
00007ffb`ad00eb74 c3              ret
00007ffb`ad00eb75 cd2e            int     2Eh
00007ffb`ad00eb77 c3              ret
```

4. Retrieve the SSN of **NtQueryInformationProcess**. Inside the exception handler, update **RAX** with NtQueryInformationProcess' SSN. I.e., the original SSN was replaced.

```
...<SNIP>...
uint32_t GetSSNByHash(PVOID pe, uint32_t Hash)
{
	PBYTE pBase = (PBYTE)pe;
```

```
    PIMAGE_DOS_HEADER   pImgDosHdr = (PIMAGE_DOS_HEADER)pBase;
    PIMAGE_NT_HEADERS   pImgNtHdrs = (PIMAGE_NT_HEADERS)(pBase + pImgDosHdr->e_lfanew);
    IMAGE_OPTIONAL_HEADER   ImgOptHdr = pImgNtHdrs->OptionalHeader;
    DWORD exportdirectory_foa = RvaToFileOffset(pImgNtHdrs,
ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
    PIMAGE_EXPORT_DIRECTORY pImgExportDir = (PIMAGE_EXPORT_DIRECTORY)(pBase + exportdirectory_foa);
//Calculate corresponding offset
    PDWORD FunctionNameArray = (PDWORD)(pBase + RvaToFileOffset(pImgNtHdrs, pImgExportDir-
>AddressOfNames));
    PDWORD FunctionAddressArray = (PDWORD)(pBase + RvaToFileOffset(pImgNtHdrs, pImgExportDir-
>AddressOfFunctions));
    PWORD  FunctionOrdinalArray = (PWORD)(pBase + RvaToFileOffset(pImgNtHdrs, pImgExportDir-
>AddressOfNameOrdinals));

    for (DWORD i = 0; i < pImgExportDir->NumberOfFunctions; i++)
    {
        CHAR* pFunctionName = (CHAR*)(pBase + RvaToFileOffset(pImgNtHdrs, FunctionNameArray[i]));
        DWORD Function_RVA = FunctionAddressArray[FunctionOrdinalArray[i]];
        if (Hash == ROR13Hash(pFunctionName))
        {
            void *ptr = malloc(10);
            if (ptr == NULL) {
                perror("malloc failed");
                return -1;
            }
            unsigned char byteAtOffset5 = *((unsigned char*)(pBase + RvaToFileOffset(pImgNtHdrs, Function_RVA)) + 4);
            //printf("Syscall number of function %s is: 0x%x\n", pFunctionName,byteAtOffset5);    //0x18
            free(ptr);
            return byteAtOffset5;
        }
    }
    return 0x0;
}
...<SNIP>...
```

5. Since we called NtDrawText but with NtQueryInformationProcess' arguments, the call should be failed. However, since we changed the SSN, the syscall is successful.

```
   fnNtQueryInformationProcess pNTQIP = (fnNtQueryInformationProcess)pNTDT;
   NTSTATUS status = pNTQIP(pi.hProcess, ProcessBasicInformation, &pbi, sizeof(PROCESS_BASIC_INFORMATION),
 NULL);
```

In this example, NtDrawtext's SSN is 0xdd, NtQueryInformationProcess' SSN is 0x19, the address of NtDrawText is 0x00007FFBAD00EB60

The call is made to NtDrawText's address but with NtQueryInformationProcess arguments. Since the SSN is changed from 0xdd to 0x19, the syscall is successful.

image.png Image not found or type unknown

Let's modify the code and play with NtDelayExecution again, considering it is easier for us to observe the call stack. As expected, these Yara rules we used before cannot detect any byte sequence pattern.

image.png Image not found or type unknown

Inspect the call stack, the syscall is initiated from the ntdll memory space, it looks legitimate regarding this aspect. However, **KeDelayExecutionThread** expects **NtDelayExecution** as the corresponding NTAPI, not NtDrawText. This clue could be used as a detection rule.

image.png Image not found or type unknown

Advantages and Detection

MutationGate has its advantages, while it is possible to detect it. If you can think of any other advantages and detections, please let me know :)

Advantages

1. Do not load the 2nd ntdll module
2. No modification to the loaded ntdll module
3. No custom syscall stub and byte sequence pattern
4. syscall is initiated in the ntdll module, which looks legitimate

Possible Detections

1. The **AddVectoredExceptionHandler** call could look suspicious in a normal program
2. The function in ntoskrnl.exe is not consistent with the one in the ntdll module
3. Initiated syscall in a benign NTAPI does not expect a different NTAPI's SSN

Comparison with Other Similar Approaches

HWSyscall(https://github.com/Dec0ne/HWSyscalls) and **TamperingSyscall**(https://github.com/rad9800/TamperingSyscalls) both ingeniously utilize hardware breakpoints to bypass inline hooks, and they are both excellent approaches. Even though I had not read and referenced these 2 projects during the period when I was inspired by and released MutationGate (after I released MutationGate, a friend sent me the links to these 2 projects), there indeed are some similar techniques or general ideas. I have carefully read and researched them, and I have summarized and compared them in a table format, as shown below.

Approach	Call	Arguments	SSN	Syscall
MutationGate	Benign NTAPI	Intended NTAPI's	Benign NTAPI's SSN -> Intended NTAPI's SSN	In the benign NTAPI
HWSyscall	Intended NTAPI	Intended NTAPI's	Intended NTAPI's SSN after retrieving it	In the closest unhooked NTAPI
TamperingSyscall	Intended NTAPI	Dummy arguments -> Intended NTAPI's	Intended NTAPI's SSN after passing EDR's check	In the intended NTAPI
Indirect Syscall	Custom ASM function	Intended NTAPI's	Intended NTAPI's SSN after retrieving it	In any unhooked NTAPI

Credits and References

During the period after I was inspired, implemented, and released MutationGate, the following resources were very helpful to me, and I am thankful to the authors.

https://github.com/senzee1984/MutationGate

https://klezvirus.github.io/RedTeaming/AV_Evasion/NoSysWhisper/

https://github.com/jthuraisamy/SysWhispers2

https://ethicalchaos.dev/2022/04/17/in-process-patchless-amsi-bypass/

https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/

https://thewover.github.io/Dynamic-Invoke/

https://cyberwarfare.live/bypassing-av-edr-hooks-via-vectored-syscall-poc/

https://redops.at/en/blog/syscalls-via-vectored-exception-handling

https://gist.github.com/CCob/fe3b63d80890fafeca982f76c8a3efdf

https://malwaretech.com/2023/12/silly-edr-bypasses-and-where-to-find-them.html

https://github.com/Dec0ne/HWSyscalls

https://github.com/rad9800/TamperingSyscalls

https://github.com/RedTeamOperations/VEH-PoC

Maldev Course

ZeroPoint Security RTO II Course

Revision #10
Created 28 February 2024 18:31:24 by winslow
Updated 23 March 2024 22:30:27 by winslow