

HookChain, novo ponto de vista para bypass de solução de EDR

Helvio Benedito Dias de Carvalho Junior
Sec4US

Abstract: No atual ecossistema de segurança digital, onde ameaças evoluem com rapidez e complexidade, as empresas desenvolvedoras de soluções de detecção e resposta em endpoints (EDR) estão em constante busca por inovações que não apenas acompanhem, mas antecipem os vetores de ataque emergentes. Nesse contexto, este artigo introduz o *HookChain*, um olhar sob outro ponto de vista de técnicas amplamente conhecidas, porém quando combinadas proporciona uma camada adicional de evasão sofisticada frente aos tradicionais sistemas EDR.

Através de uma combinação precisa de técnicas de IAT Hooking, resolução dinâmica de SSNs e chamadas de sistema indiretas, o *HookChain* redireciona o fluxo de execução dos subsistemas do Windows de maneira que permanece invisível aos olhares vigilantes dos EDRs que atuam somente na Ntdll.dll, sem necessitar de alterações no código-fonte das aplicações e malwares envolvidos.

Este trabalho não apenas desafia as convenções atuais em segurança cibernética, mas também ilumina um caminho promissor para futuras estratégias de proteção, alavancando a compreensão de que a evolução contínua é fundamental para a eficácia da segurança digital.

Ao desenvolver e explorar a técnica *HookChain*, este estudo contribui significativamente para o corpo de conhecimento em segurança de endpoints, estimulando o desenvolvimento de soluções mais robustas e adaptativas que possam efetivamente enfrentar a dinâmica sempre em transformação das ameaças digitais. Este trabalho aspira a inspirar uma reflexão profunda e o avanço na pesquisa e desenvolvimento de tecnologias de segurança que estejam sempre vários passos à frente dos adversários.

Keywords: HookChain, Bypass EDR, Evading EDR

Repositório: <https://github.com/helviojunior/hookchain/>

ACM Reference Format: Helvio Carvalho Junior. 2024. HookChain: A new perspective for Bypassing EDR Solutions. Curitiba, PR, BRAZIL. <https://arxiv.org/abs/2404.16856>

1. INTRODUÇÃO

No atual cenário corporativo, onde a segurança digital é mais crítica do que nunca, os sistemas de detecção e resposta em endpoints (EDR) emergiram como pilares essenciais na defesa contra ataques e ameaças digitais cada vez mais complexos. À medida que o mundo tecnológico se torna cada vez mais intrincado e as ameaças digitais evoluem com rapidez impressionante, as empresas têm sido impelidas a desenvolver suas próprias soluções EDR, movimentando bilhões de dólares neste mercado vibrante.

Neste estudo, destaco o novo olhar que o HookChain traz sobre as técnicas avançadas de evasão de segurança, ao escapar habilmente dos mecanismos de monitoramento e controle implementados pelos EDRs no espaço de usuário, especificamente na biblioteca Ntdll.dll. Essa biblioteca serve como um ponto crítico de coleta de telemetria para a maioria dos EDRs, operando na última fronteira antes de acessar o núcleo do sistema operacional (kernel-land – ring 0).

Através de um método sofisticado que combina o IAT Hooking (um tipo de interceptação de chamadas de função através da manipulação da tabela de importações) com a resolução dinâmica dos números de serviço do sistema (SSN) e chamadas de sistema indiretas (Indirect Syscalls), o HookChain é capaz de redirecionar o fluxo de execução de todos os principais subsistemas do Windows, como kernel32.dll, kernelbase.dll e user32.dll. Isso significa que, uma vez implantado, o HookChain garante que todas as chamadas de API dentro do contexto de uma aplicação sejam realizadas de maneira transparente, evitando completamente a detecção pelos EDRs.

O diferencial é que essa técnica é executada sem exigir qualquer modificação no código-fonte da aplicação ou malwares a ser executado, garantindo, no momento da elaboração e publicação desta pesquisa, uma evasão completa dos mecanismos de monitoramento da Ntdll.dll instalados pela grande parte dos sistemas de EDR. Esta metodologia abre novos caminhos para o desenvolvimento de estratégias de segurança mais robustas, desafiando as empresas a repensarem a eficácia de seus sistemas de proteção digital.

1.1. Objetivo e limitações

Este estudo tem como objetivo demonstrar uma nova técnica de bypass com a utilização de interceptação das funções das APIs do sistema operacional Microsoft Windows® 64 bits em espaço de usuário.

Desta forma os conceitos demonstrados são referentes ao sistema operacional Windows com o processo 64 bits executando em espaço de

usuário, desta forma não iremos nos aprofundar em outros sistemas operacionais, nem tampouco em outras arquiteturas. Bem como não aprofundaremos em outras metodologias de telemetria e by-passes como: análise estática, kernel driver, interceptações em espaço de kernel entre outras.

1.2. Ética

Este estudo não representa infrações éticas, pois todos os testes foram realizados em ambientes controlados e com licenciamento válido. Nem tampouco visa classificar os produtos de defesa e EDR aqui demonstrados quanto a sua efetividade, eficácia e qualidade no processo de proteção e defesa dos ativos onde estão instalados por se tratar de um estudo e apresentação de uma técnica focada em um único ponto de identificação dos agentes.

2. BACKGROUND

2.1. Arquitetura dos EDR

2.1.1. Overview

EDR é o acrônimo do inglês **E**ndpoint **D**etection and **R**esponse, em sua tradução livre, Agente de Detecção e Resposta, tendo como sua principal função a identificação, contenção e alerta de comportamentos maliciosos.

Um agente EDR é uma coleção de componentes de software que cria, consome, processa e transmite dados das atividades do sistema operacional para uma central, cujo seu trabalho é determinar a intenção do ator/usuário (se a intenção e comportamento é malicioso ou não) [1, p. XIX].

2.1.2. Agent Design

De forma básica os agentes são compostos por diversos componentes, de forma que cada um tem a sua função e tipo de dados capaz de coletar para a telemetria. [1, p. 9]

Os agentes/módulos mais comuns são:

- **Scanner estático:** Realiza análises estáticas dos arquivos/imagens como o PE (Portable Executable).
- **DLL Hook:** Hooking (ou interceptação) é o processo de redirecionamento do fluxo de execução da aplicação com o objetivo de interceptar chamadas específicas das APIs (Application Programming Interface) do sistema operacional.
- **Driver de Kernel:** O driver de kernel é o componente responsável por injetar o código (geralmente uma DLL) que realizará a interceptação das chamadas das funções no processo alvo. Em algumas soluções de EDR o driver de kernel também é utilizado para realizar a interceptação das chamadas de APIs a nível de Kernel.
- **Serviço de agente:** É o módulo/aplicação responsável por agregar a telemetria e eventos gerados pelos componentes do EDR, em alguns casos correlacionar estes dados, gerar alertas ou contenções. Posteriormente sincronizar estes dados com a central de gestão do EDR.

A Figura 1 ilustra estes componentes e a correlação de cada um deles. Como podemos observar, um agente EDR não se utiliza de muitas fontes de informação para tomar as suas decisões. Vale ressaltar que a quantidade de informações, forma de utilização dos módulos e posicionamento dos módulos pode variar de produto para produto.

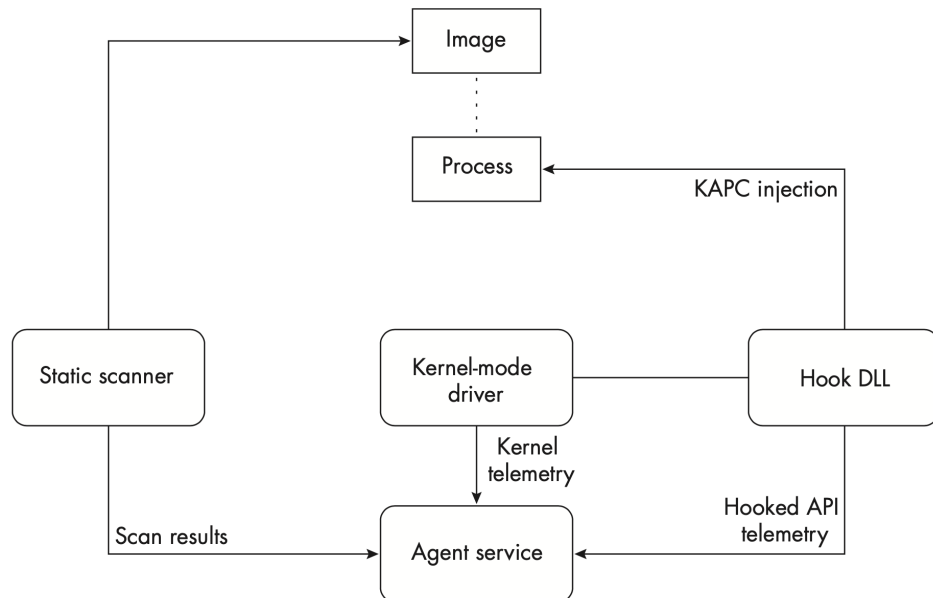


Figura 1: Arquitetura básica do agente [1, p. 10]

2.2. Windows Internals

2.2.1. Conceitos e fundamentos

2.2.1.1. Windows API

API é um acrônimo para **A**pplication **P**rogramming **I**nterface (Interface de programação de Aplicativos). API é um conjunto de métodos de comunicação entre vários componentes de software.

“A interface de programação do sistema é uma interface de programação em espaço de memória de usuário (user-land)” [2, p. 2]. Na prática tudo que realizamos no Windows (abrir arquivo, acesso de leitura ou escrita em arquivos, acessar a rede, entre outros) são realizados através das APIs do Windows. O mesmo ocorre em outros sistemas (incluindo sistemas operacionais como Linux, iOS, Android entre outros).

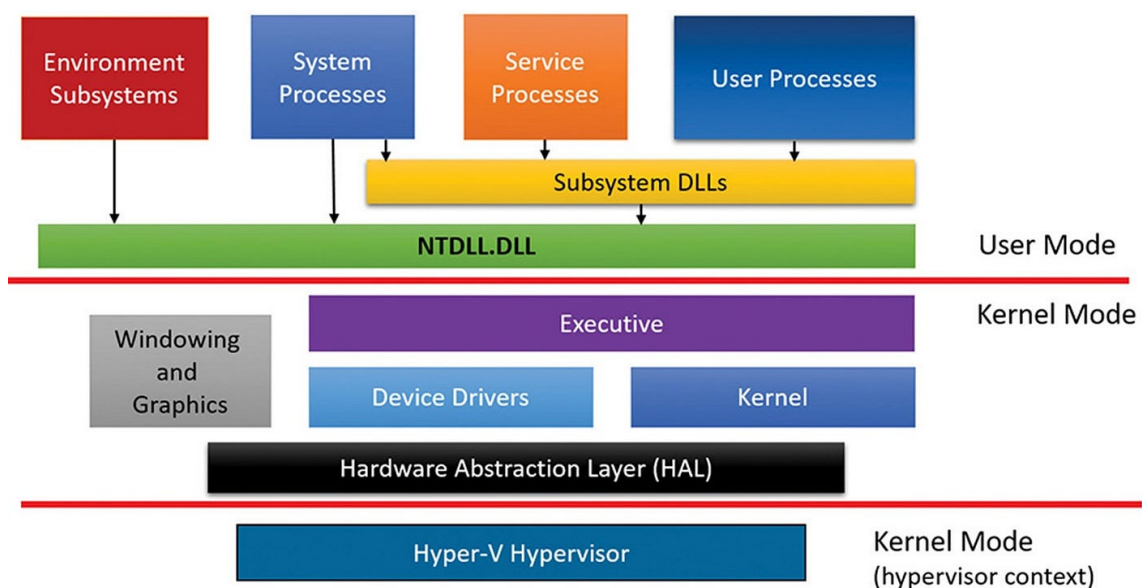


Figura 2: Arquitetura simplificada do Windows [3, p. 47]

Na Figura 2 se pode observar uma linha divisória entre os componentes residentes no modo de usuário (user mode) e os residentes no modo de kernel (Kernel mode). Bem como uma segunda linha divisória entre o modo de kernel e o hypervisor. De forma geral o hypervisor continua rodando com os mesmos privilégios do kernel (ring 0), porém como o hypervisor utiliza instruções da CPU especializadas (VT-x no Intel, SVM em AMD), ele pode se isolar do kernel enquanto monitora o mesmo (e as aplicações). [3, p. 47]

Para o objetivo deste estudo focaremos apenas no processo de transição entre o modo de usuário e modo de kernel.

2.2.2. Kernel Mode vs. User Mode

Com o objetivo de proteger as aplicações de acessarem e modificarem dados críticos do sistema operacional, o Windows utiliza dois modos de acesso (privilégios): *user mode* e *kernel mode*. Aplicações de usuário são executadas em espaço de usuário (user mode), enquanto códigos do sistema operacional como serviços do sistema e drivers de dispositivos rodam em espaço de kernel (kernel mode). [2, p. 17]

Nota: As arquiteturas x86 e AMD64 (x64) definem quatro níveis de privilégios (protection rings) com o objetivo de proteger o código e dados do sistema de alterações errôneas ou maliciosas vindas do código de menor privilégio. O Windows utiliza somente o privilégio 0 (ou ring 0) para espaço de kernel e o privilégio 3 (ou ring 3) para espaço de usuário. [2, p. 17]

2.2.3. Serviços, Funções e Rotinas

Com o objetivo de padronizar o entendimento de alguns termos neste artigo iremos utilizar as definições descritas por Russinovich [2, p. 4]

- **DLLs (dynamic-link libraries):** Um pacote com diversas funções disponíveis para utilização. Exemplos: Kernel32.dll, User32.dll e ntdll.dll.
- **Funções da API Windows (Windows API functions):** Sub-rotinas/funções documentadas e disponíveis para utilização (em modo de usuário) nas APIs do Windows. Exemplos: CreateProcess e CreateFile da DLL Kernel32.dll e GetMessage da DLL User32.dll.
- **Serviços de sistema nativos (Native system services):** Ou também conhecido como System Calls, são funções não documentadas e disponíveis para utilização (em modo de usuário). Estas funções estão presentes dentro da DLL ntdll.dll e apresentam sua nomenclatura iniciada em **Nt** ou **Zw**. Por exemplo, NtCreateUserProcess é a função interna chamada pela função CreateProcess para criar um processo.

2.2.4. System Service Dispatching

De forma básica o System Service Dispatching é o portão de transição do ring 3 (user mode) para o ring 0 (kernel mode). O System Service Dispatching é uma das interrupções capturadas pelo kernel (Kernel's trap handlers dispatch interrupts) de forma que o *system service dispatch* é o resultado de uma execução disparada por uma instrução designada para o system service. [2, p. 132]

Conforme descrito na convenção de chamada da arquitetura AMD64 o Windows utiliza a instrução assembly *syscall*, passando o *system call number* (também conhecido como SSN – System Service Number) no registrador EAX bem como os 4 primeiros parâmetros da função em registradores e todos os outros parâmetros (quando aplicável) na pilha. [4]

Por questões de segurança a Microsoft altera o SSN de cada função quando um novo Service Pack ou Release do Windows é lançado. Eventualmente novas funções podem ser adicionadas ou removidas.

Nota: Como veremos mais a frente, este processo de randomização do SSN faz com que tenhamos que resolvê-lo de forma dinâmica para a correta utilização das funções de forma direta.

Como na arquitetura de 64 bits existe somente um mecanismo para execução das chamadas de sistema (System calls), o ponto de entrada do *system service* na *ntdll.dll* utiliza a instrução *syscall* diretamente, como podemos ver abaixo:

```
0:002> u ntdll!NtReadFile
ntdll!NtReadFile:
00007ffe`b258d090 4c8bd1          mov     r10,rcx
00007ffe`b258d093 b806000000      mov     eax,6
...
00007ffe`b258d0a2 0f05           syscall
00007ffe`b258d0a4 c3             ret
```

Adicionalmente podemos observar que o valor 6 foi atribuído no registrador EAX, de forma que nesta release/service pack do windows o SSN da função **NtReadFile** é o decimal 6.

Conforme podemos ver na Figura 3, após a transição para o kernel mode, o SSN é utilizado para localizar o respectivo serviço.

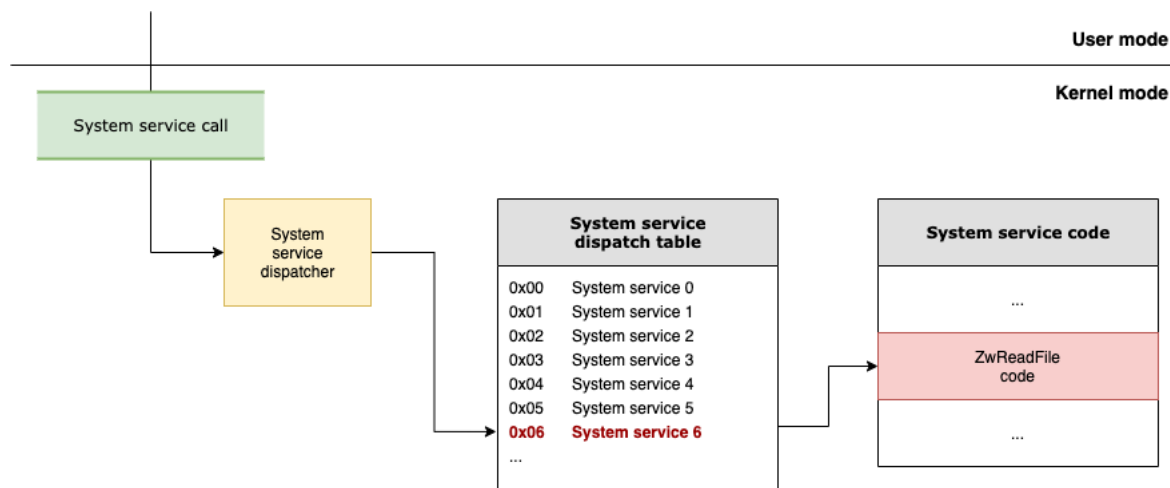


Figura 3: System service exceptions [2, p. 135]

Adicionalmente podemos verificar no código da função `ZwReadFile`, em kernel mode, que o SSN utilizado é exatamente o mesmo.

```
lkd> uf nt!ZwReadFile
nt!ZwReadFile:
fffff806`0e7f9e00 488bc4      mov     rax, rsp
fffff806`0e7f9e03 fa          cli
fffff806`0e7f9e04 4883ec10    sub     rsp, 10h
fffff806`0e7f9e08 50          push    rax
fffff806`0e7f9e09 9c          pushfq
fffff806`0e7f9e0a 6a10        push    10h
fffff806`0e7f9e0c 488d052d880000 lea     rax, [nt!KiServiceLinkage (fffff806`0e802640)]
fffff806`0e7f9e13 50          push    rax
fffff806`0e7f9e14 b806000000  mov     eax, 6
fffff806`0e7f9e19 e9e2710100  jmp     nt!KiServiceInternal (fffff806`0e811000)
```

A Figura 4 resume este processo, na arquitetura AMD64, ilustrando todo o caminho da chamada iniciando na função `WriteFile` da `Kernel32.dll` que por sua vez irá importar e executar a função `WriteFile` da `Kernelbase.dll`, que após algumas verificações de parâmetros irá realizar a chamada da função `NtWriteFile` na `ntdll.dll`, onde será executado a correta chamada da instrução `syscall`, passando o SSN que representa a função `NtWriteFile`. O system service dispatcher (função `KiSystemService` no `Ntoskrnl.exe`) então irá executar a real implementação da função `NtWriteFile`.



Figura 4: System service dispatching [2, p. 138]

2.2.5. Image Loader

Quando um processo é iniciado diversas ações são realizadas internamente pelo sistema operacional, algumas em modo de usuário e outras em modo de kernel. Para o objetivo deste estudo focaremos no processo de resolução das DLLs referenciadas e importação/referência das funções.

O *image loader* é um código residente no modo de usuário, dentro da `Ntdll.dll` e não em uma biblioteca do kernel. Desta forma há uma garantia de que a `Ntdll.dll` sempre estará presente no processo em execução (`Ntdll.dll` sempre é carregada). [2, p. 232]

Os executáveis e DLLs seguem um formato conhecido como Portable Executable (PE) e COFF (Common Object File Format) respectivamente. O nome "Portable Executable" refere-se ao fato de que o formato não é específico da arquitetura. [5]

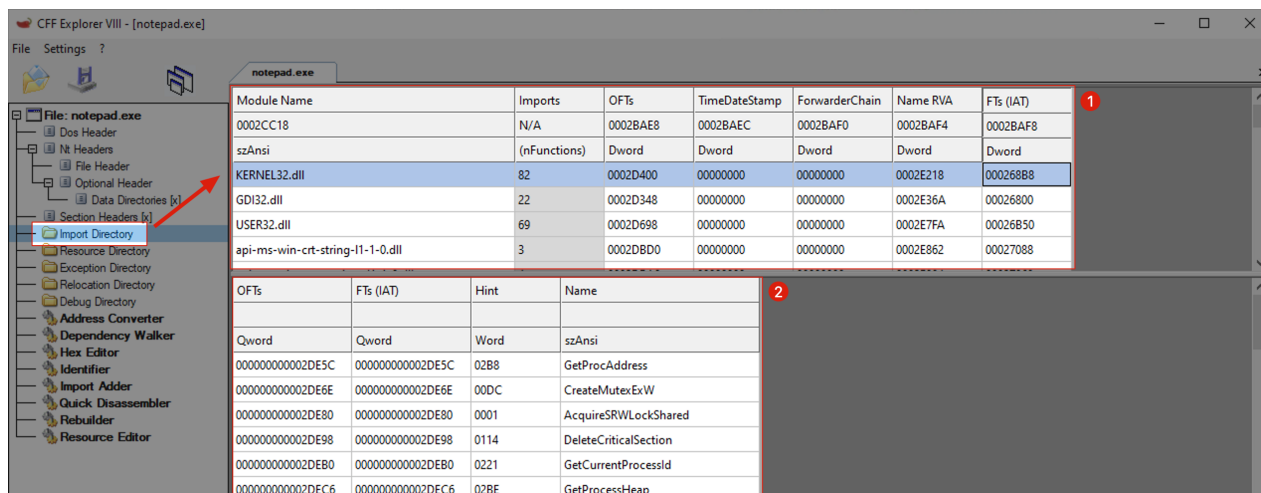


Figura 5: Import Directory do notepad.exe

Na Figura 5 se pode observar a utilização do software CFF Explorer [6] para visualizar a tabela de importação (Import Directory), onde são definidas todas as DLLs referenciadas pela aplicação, bem como as funções referenciadas de cada DLL.

Durante o carregamento da aplicação uma outra tabela chamada IAT (Import Address Table) é preenchida com os endereços atuais da função em memória. Este processo é realizado de forma dinamica para cumprir diversos requisitos como realocação de memória, ASLR Address space layout randomization entre outros.

2.2.5.1. IAT - Import Address Table

Durante o processo de inicialização e carregamento de uma aplicação a IAT é preenchida com o endereço atual de cada função referenciada pela aplicação. Para este processo os seguintes passos são realizados:

1. Carrega cada uma por uma das DLLs referenciadas na tabela de importação do PE.
2. Verifica se a DLL em questão já está carregada na memória do processo, caso não, realiza a leitura da DLL em disco e a mapeia em memória.
3. Após o mapeamento da DLL em memória, este processo é repetido para esta DLL com o objetivo de importar as dependencias utilizadas por ela.
4. Após cada DLL carregada, é realizado o tratamento da IAT buscando as funções específicas a serem importadas. Geralmente este processo é realizado pelo nome da função, porém há a possibilidade de ser realizado por um número indexador. Para cada nome importado o carregador verifica a tabela de exportação da DLL

importada e tenta localizar a função desejada. Caso não encontre esta operação é abortada.

```
0:002> lm
start                end                module name
00007ff7`3dddf0000 00007ff7`3de28000  notepad
...

0:002> !dh 00007ff7`3dddf0000 -f

File Type: EXECUTABLE IMAGE
...
    0 [      0] address [size] of Export Directory
2D0E8 [  244] address [size] of Import Directory
36000 [   BD8] address [size] of Resource Directory
33000 [  10E0] address [size] of Exception Directory
    0 [      0] address [size] of Security Directory
37000 [   2D8] address [size] of Base Relocation Directory
2AC40 [    54] address [size] of Debug Directory
    0 [      0] address [size] of Description Directory
    0 [      0] address [size] of Special Directory
    0 [      0] address [size] of Thread Storage Directory
266D0 [   118] address [size] of Load Configuration Directory
    0 [      0] address [size] of Bound Import Directory
267E8 [   900] address [size] of Import Address Table Directory
2CA00 [    E0] address [size] of Delay Import Directory
    0 [      0] address [size] of COR20 Header Directory
    0 [      0] address [size] of Reserved Directory

0:002> dps 00007ff7`3dddf0000+267E8 00007ff7`3dddf0000+267E8+900
00007ff7`3de167e8 00007ffe`a1b86980 COMCTL32!CreateStatusWindowW
00007ff7`3de167f0 00007ffe`a1b32ac0 COMCTL32!TaskDialogIndirect
...
00007ff7`3de168b8 00007ffe`b1b8b1d0 KERNEL32!GetProcAddressStub
00007ff7`3de168c0 00007ffe`b1b94ca0 KERNEL32!CreateMutexExW
...
```

Na saída acima se pode observar a listagem da IAT do processo notepad.exe, bem como na saída abaixo observa-se que no endereço indicado realmente é o código da função mapeada.

```
0:002> u 00007ffe`b1b8b1d0
KERNEL32!GetProcAddressStub:
00007ffe`b1b8b1d0 4c8b0424      mov     r8,qword ptr [rsp]
00007ffe`b1b8b1d4 48ff25a5580600 jmp     qword ptr [KERNEL32!_imp_GetProcAddressForCaller
(00007ffe`b1bf0a80)]
00007ffe`b1b8b1db cc          int     3
00007ffe`b1b8b1dc cc          int     3
```

A Figura 6 ilustra todo este esquema de importação do qual foi detalhado.

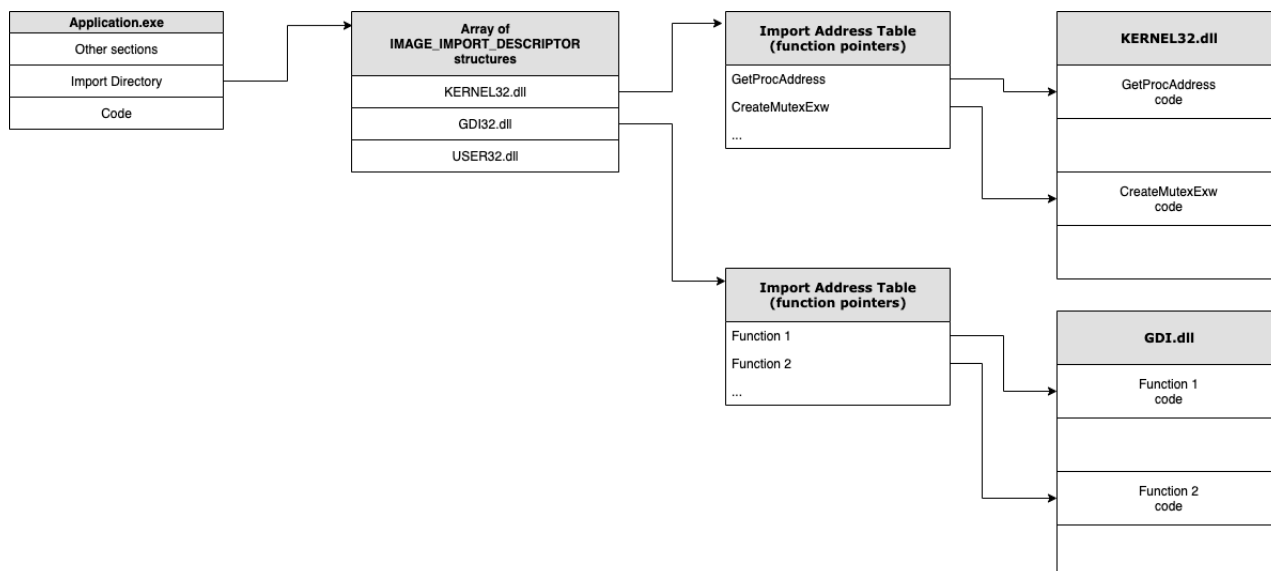


Figura 6: PE Import schema

2.3. Function Hook

A interceptação (Hook) de funções não é algo novo e tem diversas aplicações como debug de aplicações (como o implementado pelo software API Monitor [7]) mas também no processo de monitoramento pelas camadas de defesa (EDR).

A ideia geral por trás da interceptação das funções se dá no fato de se inserir no fluxo de controle da aplicação a ser monitorada. O agente de monitoramento assume o controle da função monitorada antes de que o código original seja executado, após a análise desejada (podendo ser log, telemetria, controle entre outros) o fluxo da execução é transferido para a função original. [8, p. 687]

Para a realização deste processo existem diversas abordagens disponíveis, neste artigo iremos tratar das mais utilizadas pelos EDRs: 1 – Utilização de **JMP** ou **CALL**; 2 – **Manipulação da IAT** (Import Address Table). Em ambas as estratégias o EDR realiza as manipulações desejadas em tempo de execução, ou seja, no momento do carregamento da aplicação o EDR recebe o evento e realiza a injeção da sua DLL de Hook, que por sua vez irá alterar o código desejado da aplicação algo (a ser monitorada).

Nota: Conforme descrito anteriormente, os fluxos e diagramas são referentes ao Windows 64 bits com a aplicação igualmente executando em 64 bits.

2.3.1. JMP ou CALL

Esta estratégia geralmente é utilizada para alterar o código das chamadas das funções nativas dentro da ntdll.dll.

Podemos ver no código abaixo a função NtCreateProcess original, ou seja, sem a presença de um hook.

```
0:002> u ntdll!NtCreateProcess
ntdll!NtCreateProcess:
00007ffe`b258e700 4c8bd1      mov     r10,rcx
00007ffe`b258e703 b8ba000000    mov     eax,0BAh
...
00007ffe`b258e712 0f05        syscall
00007ffe`b258e714 c3          ret
00007ffe`b258e715 cd2e        int     2Eh
00007ffe`b258e717 c3          ret
```

Agora quando se tem a presença de um EDR pode-se observar que as primeiras instruções são substituídas por um JMP (podendo ser um CALL, porém é menos comum de se ver), redirecionando então, o fluxo de execução da aplicação para um código arbitrário injetado pelo EDR.

```
0:004> u ntdll!NtCreateProcess
ntdll!NtCreateProcess:
00007fff`96bee700 e9f81b1600    jmp     00007fff`96d502fd
00007fff`96bee705 cc            int     3
00007fff`96bee706 cc            int     3
00007fff`96bee707 cc            int     3
...
00007fff`96bee712 0f05        syscall
00007fff`96bee714 c3          ret
```

E o endereço de destino do JMP não está atrelado a nenhum módulo (DLL) conhecido, sendo assim um código injetado em tempo de execução.

```
0:004> !address 00007fff`96d502fd

Usage:                <unknown>
Base Address:         00007fff`96d50000
End Address:          00007fff`96d53000
Region Size:          00000000`00003000 ( 12.000 kB)
State:                00001000          MEM_COMMIT
Protect:              00000020          PAGE_EXECUTE_READ
Type:                 00020000          MEM_PRIVATE
Allocation Base:      00007fff`96d50000
Allocation Protect:    00000002          PAGE_READONLY

Content source: 1 (target), length: 2d03
```

2.3.2. IAT Hook

Um dos primeiros registros do processo de interceptação de função através da manipulação da IAT foi descrito por Matt Pietrek em 1995 em seu livro Windows 95 System Programming Secrets [8, p. 687]

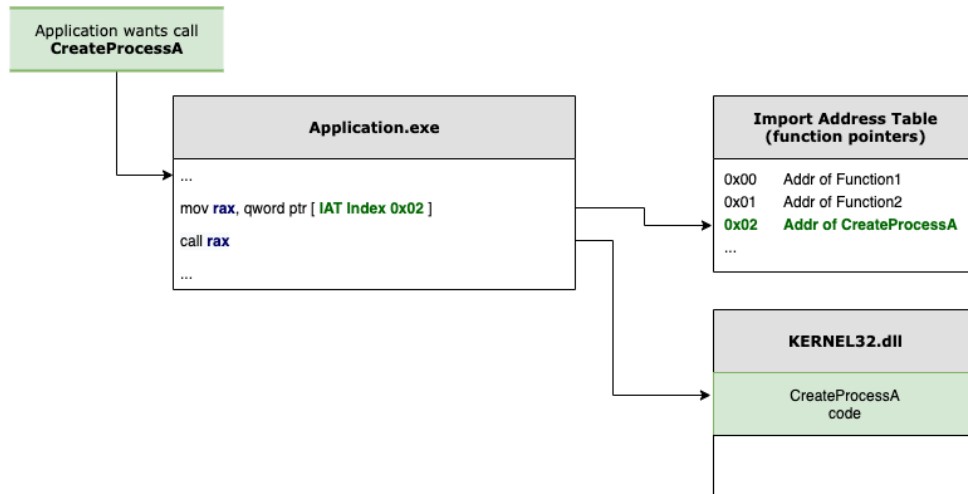


Figura 7: Fluxo de execução original

A Figura 7 demonstra o fluxo de execução original da aplicação, onde quando a aplicação necessita realizar uma chamada de função externa (referenciada em outra DLL) a aplicação busca na IAT o endereço da função desejada e posteriormente realiza o CALL para este endereço.

Em contrapartida na Figura 8 se pode observar que o endereço da função na IAT foi substituído por um endereço arbitrário (função interceptadora) que opcionalmente pode realizar a execução da função original.

Nos cenários onde essa interceptação é realizada pelo EDR o endereço presente na IAT será o endereço da função do EDR que fará os processos de telemetria, checagens, logs e demais trabalhos previstos pelo EDR.

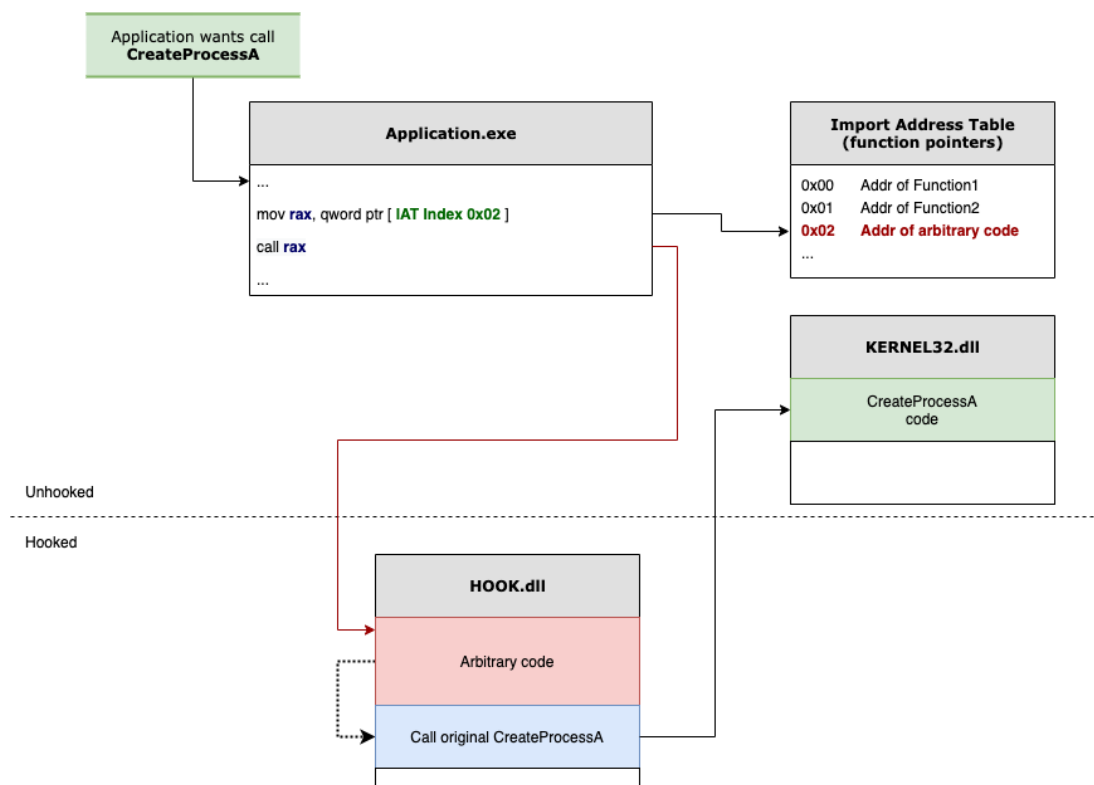


Figura 8: Fluxo de execução com interceptação

2.4. Bypasses conhecidos

No tocante ao bypass dos hooks realizados pelo EDR existem diversas técnicas possíveis e publicamente divulgadas, porém comumente elas se reduzem as seguintes técnicas:

- Remapeamento da Ntdll.dll para obter o código original ou sobrescrever o código das funções na área de memória previamente mapeada.
- Chamadas das syscalls diretamente (*direct syscalls*).

Grande parte dos EDRs atualmente presentes no mercado centralizam seu ponto de monitoramento, em espaço de usuário, através da interceptação das chamadas na Ntdll.dll através da técnica de JMP, sendo assim as técnicas de bypass dos hooks em modo de usuário publicamente reportadas até o momento atuam em torno da Ntdll.dll.

2.4.1. Remapeamento da Ntdll.dll

A técnica de remapeamento da ntdll, assim como outras técnicas, pode haver diversas variantes. De forma geral o remapeamento consiste em ler uma cópia íntegra da ntdll.dll (sem os hooks), geralmente diretamente do disco, e posteriormente sobrescrever a área de memória relativa as funções interceptadas.

Outra forma comum de obter uma cópia da ntdll.dll sem as interceptações, é a criação de um processo em modo suspenso, e posterior leitura da ntdll.dll deste processo, pois como vimos anteriormente a ntdll.dll é imprescindível e crucial para o carregamento e execução de um novo processo, sendo assim mesmo em modo suspenso o processo já detém em sua área de memória uma cópia da ntdll.dll, e como o processo de carregamento do processo ainda não foi finalizado, o EDR ainda não recebeu o call-back para injetar sua DLL de Hook, sendo assim a cópia da ntdll.dll neste processo ainda está íntegra (sem os hooks).

2.4.2. Direct Syscalls

De longe, a metodologia de evasão dos hooks inseridos nas funções da Ntdll.dll é a realização das chamadas diretas da Syscall. [1, p. 25]

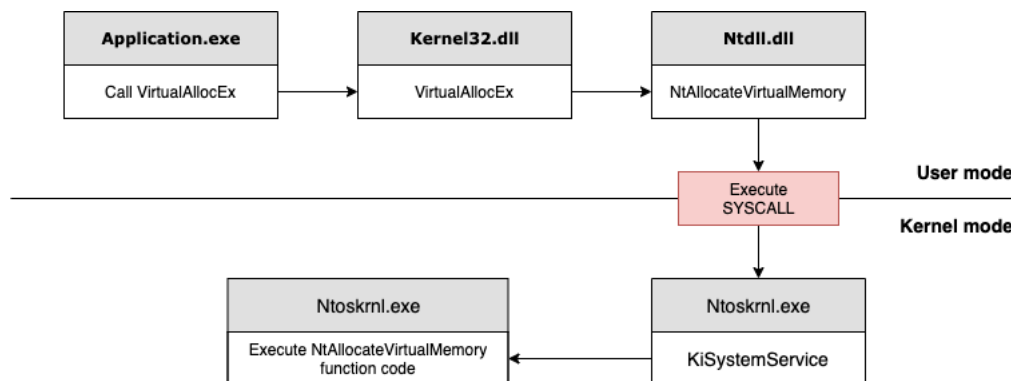


Figura 9: Fluxo de execução normal da NtAllocateVirtualMemory

A Figura 9 demonstra o fluxo normal e esperado para uma aplicação realizar a chamada da função NtAllocateVirtualMemory na Ntdll.dll.

A realização deste processo consiste em reconstruir o código da função desejada da Ntdll.dll conforme no exemplo abaixo:

```

NtAllocateVirtualMemory PROC
    mov r10, rcx
    mov eax, <SSN>
    syscall
    ret
NtAllocateVirtualMemory ENDP
  
```

Posteriormente na aplicação C++ criar a definição da função:

```

EXTERN_C NTSTATUS NtAllocateVirtualMemory(
    HANDLE    ProcessHandle,
    PVOID     BaseAddress,
    ULONG     ZeroBits,
    PULONG    RegionSize,
    ULONG     AllocationType,
    ULONG     Protect );
  
```

Desta forma a aplicação executa de forma direta a instrução SYSCALL [9] sem passar por nenhuma das DLLs de subsistemas do Windows (User32.dll, kernel32.dll entre outras) nem tampouco pela Ntdll.dll, conforme ilustrado na Figura 10.

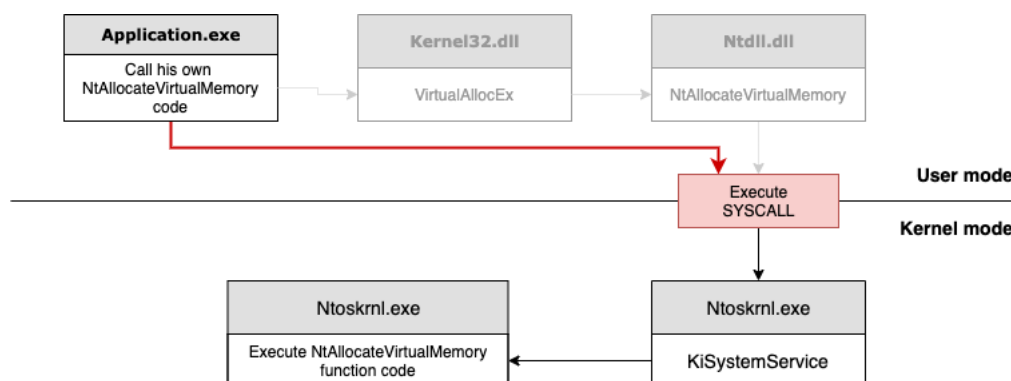


Figura 10: Fluxo de execução direto da NtAllocateVirtualMemory

Esta metodologia tem a vantagem de evadir todos os hooks em modo do usuário pois todo o controle da execução está na própria aplicação. Porém

há uma alta probabilidade de identificação pelo EDR em virtude de algumas telemetrias como:

- Tempo de execução total do processo.
- Cadeia de execução, onde o EDR espera que a chamada da função tenha vindo da aplicação, posteriormente passado pela Kernel32.dll, posteriormente pela Ntdll.dll.

Além da possibilidade de identificação, há outros pontos negativos desta metodologia:

- Necessidade de mapeamento manual de cada SSN (System Service Number) e sua função correlata, pois como vimos anteriormente o Windows altera estes números a qualquer momento sem nenhum aviso prévio.
- Um grande esforço de programação para portar os códigos desejados que utilizam as DLLs de subsistemas do Windows para utilizar somente as funções nativas através da chamada direta Syscall.
- Baixa portabilidade de códigos pré-existentes. Pois há a necessidade de ajustar o código fonte da aplicação para utilizar somente chamadas nativas como as Nt... e Zw...

2.4.3. Indirect Syscall

Uma variante bastante utilizada da técnica acima é a *Indirect Syscall*, que consiste na alteração da função para ao invés de executar a instrução SYSCALL diretamente, realizar um JMP para um endereço de memória dentro da Ntdll.dll que contenha a instrução Syscall.

Considerando o código abaixo (extraído de uma função qualquer) da Ntdll.dll

```
0:002> u ntdll!NtCreateProcess
ntdll!NtCreateProcess:
00007ffe`b258e700 4c8bd1      mov     r10,rcx
00007ffe`b258e703 b8ba000000  mov     eax,0BAh
...
00007ffe`b258e712 0f05        syscall
00007ffe`b258e714 c3          ret
00007ffe`b258e715 cd2e        int     2Eh
00007ffe`b258e717 c3          ret
```

Se pode alterar a réplica da função para após a definição do EAX realizar um JMP para o endereço da instrução SYSCALL.

```
NtAllocateVirtualMemory PROC
    mov r10, rcx
    mov eax, <SSN>
    JMP 00007ffe`b258e712
    ret
NtAllocateVirtualMemory ENDP
```

Esta pequena alteração traz uma grande efetividade em virtude de do ponto de vista da telemetria de cadeia de execução a chamada da instrução syscall terá vindo da Ntdll.dll e não mais do código direto da aplicação.

2.4.4. Resolução dinâmica do SSN

Em dezembro de 2020 o @modexpblog descreveu em seu blog um post nomeado “Bypassing User-Mode Hooks and Direct Invocation of System Calls for Red Teams” [9] onde descreve detalhadamente como realizar a correlação dinâmica entre o Número do Syscall (SSN) e a sua função atrelada, desta forma o bypass fica mais confiável, uma vez que não necessita conter uma lista dos SSNs para cada build do Windows fixada dentro da aplicação. Essa técnica utiliza o seguinte fluxo:

1. Localiza o endereço base da Ntdll.dll utilizando as tabelas TEB (Thread Environment Block) e PEB (Process Environment Block).
2. Enumera todas as funções com o nome iniciado em “Zw”, pois em espaço de usuário as funções Nt... e Zw... apontam para o mesmo endereço, sendo assim não tendo uma diferença prática na utilização do Zw ou Nt neste cenário.
3. Armazena (em uma array) o endereço virtual relativo (RVA – Relative Virtual Address) e o nome das funções enumeradas no passo anterior. Na implementação deste algoritmo o autor utiliza uma técnica de evasão do EDR que consiste de ao invés de salvar e utilizar o nome da função como chave de comparação, é utilizado um hash calculado por um algoritmo próprio através de operações aritméticas com o ROR.
4. Realiza a ordenação da array pelos endereços das funções.
5. Define o SSN como sendo o indexador da array.

Esta técnica é simples e efetiva, pois, o código das funções Zw/Nt então em um único bloco de código sequencial conforme pode-se visualizar na Figura 11.

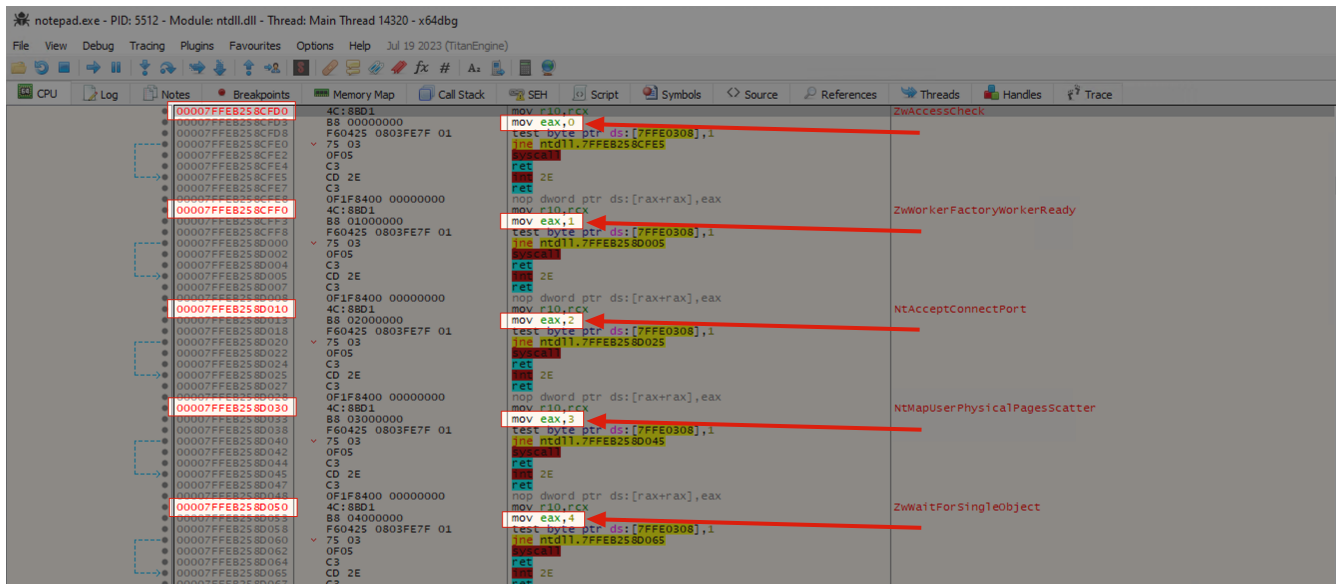


Figura 11: Funções Ntdll.dll Zw/Nt em memória e os seus respectivos SSN

2.4.5. Resolução dinâmica do SSN – Halo's Gate

Outras técnicas de resolução dinâmica foram publicadas no decorrer dos últimos anos como a *Hell's gate* [10] publicada em junho de 2020 e a *Halo's gate* publicada em abril de 2021 [11] pelo Reenz0h da Sektor7.

A *Halo's gate*, de forma geral realiza o seguinte fluxo:

1. Localiza o endereço atual da função desejada dentro da Ntdll.dll.
2. Realiza a leitura dos bytes da função (atualmente 32 bytes) e verifica se os bytes da função correspondem aos das instruções assembly (mov r10, rcx; mov eax, SSN).
3. Caso não sejam estes bytes a função está sendo monitorada (em outras palavras tem um Hook definido), porém as funções vizinhas (antes e depois) podem não estar com Hook.
4. Busca nas funções vizinhas (acima e abaixo) por funções sem um Hook, e calcula a distância da função localizada com a função atual, tendo assim o código SSN da função atual.

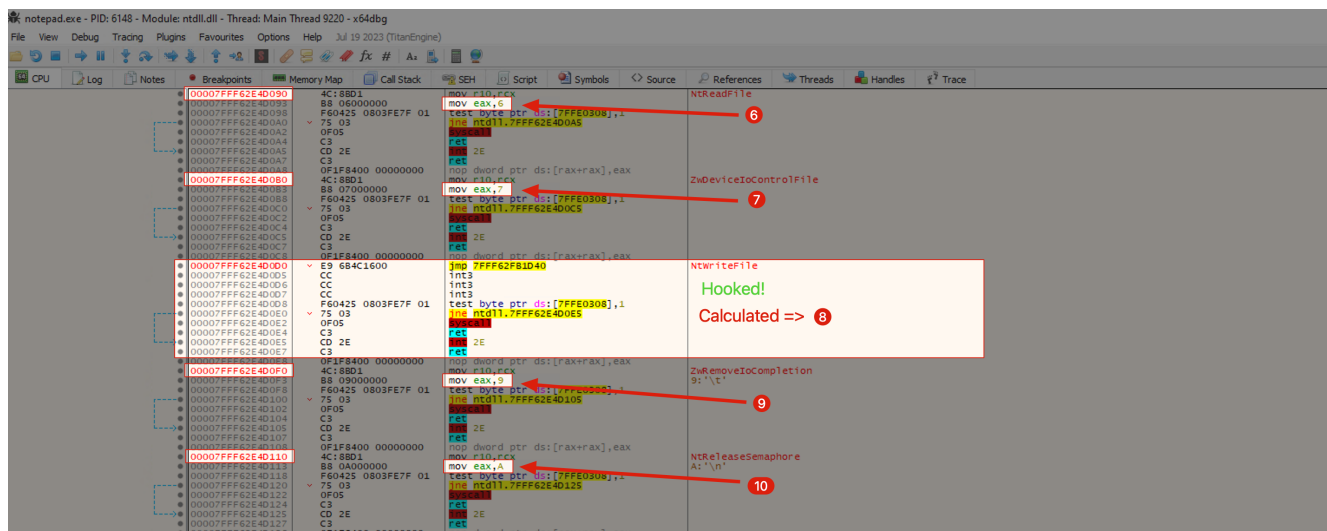


Figura 12: Funções Ntdll.dll Zw/Nt em memória e os seus respectivos SSN

A Figura 12 demonstra de forma clara um hook na função NtWriteFile, através da presença da instrução JMP ao invés do mov r10, rcx. Porém as funções vizinhas ZwDeviceIoControlFile e ZwRemoveIoCompletion não estão com hook definido e os seus SSN são 7 e 9 respectivamente. Sendo assim pode-se inferir que o SSN da função NtWriteFile é 8.

A Figura 13 exhibe trecho do código utilizado pela Halo's gate.

```
// check neighboring syscall down
if (*(PBYTE)pFunctionAddress + idx * DOWN) == 0x4c
    && *((PBYTE)pFunctionAddress + 1 + idx * DOWN) == 0x8b
    && *((PBYTE)pFunctionAddress + 2 + idx * DOWN) == 0xd1
    && *((PBYTE)pFunctionAddress + 3 + idx * DOWN) == 0xb8
    && *((PBYTE)pFunctionAddress + 6 + idx * DOWN) == 0x00
    && *((PBYTE)pFunctionAddress + 7 + idx * DOWN) == 0x00) {
    BYTE high = *((PBYTE)pFunctionAddress + 5 + idx * DOWN);
    BYTE low = *((PBYTE)pFunctionAddress + 4 + idx * DOWN);
    pVxTableEntry->wSystemCall = (high << 8) | low - idx;

    return TRUE;
}
```

Figura 13: Trecho do código de comparação da Halo's gate.

Como definido pelo próprio autor da técnica, o Halo's gate "É como uma onda em um lago - você começa do centro e se move para as bordas até encontrar um syscall limpo" [11], Em outras palavras o Halo's realiza o cálculo do número SSN olhando para os números dos vizinhos e ajustar de acordo. Se os vizinhos também estiverem com Hook, verifica os vizinhos de seus vizinhos e assim por diante.

Para maiores detalhes e prova de conceito da implementação do Halo's gate segue a referência da implementação desenvolvida pelo Caio Joca [12].

3. PRÉ ANÁLISE

3.1. Objetivo da análise

Esta análise tem por objetivo o fator decisório para a continuidade (ou não) deste estudo. Sendo assim a pré análise não teve a pretensão de validar a efetividade dos produtos testados, mas sim realizar a verificação dos pontos de interceptação (Hook) utilizados pelos produtos de EDR no tocante aos subsistemas Windows e NTDLL.

Por não se tratar de um código ofensivo, ou seja, nenhum código de bypass foi utilizado neste ponto do estudo, e tendo, mais uma vez, como foco apenas a enumeração e identificação dos pontos de interceptação dos produtos de EDR do mercado, o executável utilizado para esta enumeração foi disponibilizado para que alguns amigos o executassem em seus ambientes e me enviasse os resultados. Sendo assim não tive acesso aos ambientes bem como as configurações aplicadas em cada ambiente.

Nota: Por se tratar de apenas uma análise de viabilidade, os resultados obtidos nesta fase do estudo não são considerados nos resultados dos testes finais, sendo assim, o fato de não terem sido executados em um ambiente controlado, nem tampouco por mim, não impacta no resultado final deste estudo.

3.2. Metodologia de testes e Limitações

Para que essa enumeração fosse linear e fidedigna em todas as plataformas as seguintes premissas foram adotadas:

- Utilização de um código único e igual para todos os testes.
- Aplicação desenvolvida em C e compilada utilizando GCC em ambiente windows 64 bits.
- Inexistência de alterações e/ou recompilações durante o processo de enumeração. Provendo exatamente o mesmo PE (EXE), para execução em todos os ambientes testados. Todos executaram o mesmo EXE, tendo então o mesmo comportamento e hash para todas as soluções.
- Aplicação desenvolvida sem nenhum bypass ou ação de evasão das soluções.
- Execução da aplicação em qualquer versão do Windows com usuário não privilegiado, ou seja, sem permissão de administrador local.

3.3. Pontos analisados

O artefato (executável) desenvolvido para essa análise realiza a verificação de existência de hooks em 2 pontos distintos da aplicação: 1 –

funções da Ntdll.dll; 2 – IAT Hook.

3.3.1. Ntdll Hook

Para a validação de existência de Hooks nas funções da Ntdll foram realizados os seguintes passos:

1. Listadas todas as funções das quais os seus nomes iniciam com Zw ou Nt;
2. Verificado a presença da instrução JMP no código da função;

3.3.2. IAT Hook

Para a verificação da presença de Hooks na IAT das DLLs carregadas no processo os seguintes passos foram realizados:

1. Listado todas as DLLs carregadas no processo;
3. Verificado na IAT de todas as DLLs carregadas a referência de importação da DLL Ntdll.dll, bem como a utilização das funções das quais os seus nomes iniciam com Zw ou Nt;
2. Checagem se o endereço presente na IAT é diferente do endereço real da função na Ntdll.

3.4. Exemplo de resultados

Nos exemplos do resultado dos comandos abaixo foram suprimidas diversas linhas para otimizar a visualização neste documento, tendo a presença destes textos aqui apenas para referência e exemplo da resultante.

O executável e código utilizado nesta fase do estudo está disponível no git dessa pesquisa no commit 0b4a953 [13].

3.4.1. Sem a presença de hooks

```
[+] Listing ntdll Nt/Zw functions
-----
Mapped 478 functions

[+] Listing loaded modules
-----
C:\Users\M4v3r1ck\Desktop\hookchain_finder64.exe is loaded at 0x00007ff77bc30000.
C:\WINDOWS\SYSTEM32\ntdll.dll is loaded at 0x00007ff8ee910000.
C:\WINDOWS\System32\KERNEL32.DLL is loaded at 0x00007ff8eca90000.
C:\WINDOWS\System32\KERNELBASE.dll is loaded at 0x00007ff8ec590000.
C:\WINDOWS\SYSTEM32\apphelp.dll is loaded at 0x00007ff8e9720000.
C:\WINDOWS\System32\msvcrt.dll is loaded at 0x00007ff8ee290000.

[+] Listing hooked modules
-----
Checking ntdll.dll at KERNEL32.DLL IAT
+-- 0 hooked functions.
```

```
Checking ntdll.dll at KERNELBASE.dll IAT
+-- 0 hooked functions.

Checking ntdll.dll at apphelp.dll IAT
+-- 0 hooked functions.

Checking ntdll.dll at msvcrt.dll IAT
+-- 0 hooked functions.
```

3.4.2. Hooks presentes somente na Ntdll.dll

```
[+] Listing ntdll Nt/Zw functions
-----
NtAdjustPrivilegesToken is hooked
NtAlpcConnectPort is hooked
NtAlpcCreatePort is hooked
NtAlpcSendWaitReceivePort is hooked
NtClose is hooked
NtCommitTransaction is hooked
NtCreateMutant is hooked
NtCreateProcess is hooked
NtCreateProcessEx is hooked
NtCreateSection is hooked
NtCreateSectionEx is hooked
NtCreateThread is hooked
...
NtUnmapViewOfSection is hooked
NtWriteFile is hooked
NtWriteVirtualMemory is hooked
Mapped 478 functions

[+] Listing loaded modules
-----
C:\Users\M4v3r1ck\Desktop\hookchain_finder64.exe is loaded at 0x00007ff736e80000.
C:\WINDOWS\SYSTEM32\ntdll.dll is loaded at 0x00007ff8657d0000.
C:\WINDOWS\System32\KERNEL32.DLL is loaded at 0x00007ff865590000.
C:\WINDOWS\System32\KERNELBASE.dll is loaded at 0x00007ff8632e0000.
C:\WINDOWS\SYSTEM32\apphelp.dll is loaded at 0x00007ff8606c0000.
C:\WINDOWS\System32\msvcrt.dll is loaded at 0x00007ff864ae0000.

[+] Listing hooked modules
-----
Checking ntdll.dll at KERNEL32.DLL IAT
+-- 0 hooked functions.

Checking ntdll.dll at KERNELBASE.dll IAT
+-- 0 hooked functions.

Checking ntdll.dll at bdhkm64.dll IAT
+-- 0 hooked functions.

Checking ntdll.dll at atcuf64.dll IAT
+-- 0 hooked functions.

Checking ntdll.dll at apphelp.dll IAT
+-- 0 hooked functions.

Checking ntdll.dll at msvcrt.dll IAT
+-- 0 hooked functions.
```

3.4.3. Presença de hooks na IAT

```
[+] Listing ntdll Nt/Zw functions
-----
NtCreateThreadEx is hooked
NtCreateUserProcess is hooked
NtDuplicateObject is hooked
NtFreeVirtualMemory is hooked
NtLoadDriver is hooked
NtMapUserPhysicalPages is hooked
NtMapViewOfSection is hooked
NtOpenProcess is hooked
NtQuerySystemInformation is hooked
NtQuerySystemInformationEx is hooked
NtQuerySystemTime is hooked
NtQueueApcThread is hooked
NtQueueApcThreadEx is hooked
NtQueueApcThreadEx2 is hooked
NtReadVirtualMemory is hooked
NtResumeThread is hooked
NtSetContextThread is hooked
NtSetInformationProcess is hooked
NtSetInformationThread is hooked
NtTerminateProcess is hooked
NtUnmapViewOfSection is hooked
NtWriteVirtualMemory is hooked
Mapped 478 functions

[+] Listing loaded modules
-----
C:\Users\M4v3r1ck\Desktop\hookchain_finder64.exe is loaded at 0x00007ff770d10000.
C:\WINDOWS\SYSTEM32\ntdll.dll is loaded at 0x0000015158f10000.
C:\WINDOWS\System32\kern3132.dll is loaded at 0x0000015159110000.
C:\WINDOWS\SYSTEM32\ntdll.dll is loaded at 0x00007ff9e1290000.
C:\WINDOWS\System32\KERNEL32.DLL is loaded at 0x00007ff9e0250000.
C:\WINDOWS\System32\KERNELBASE.dll is loaded at 0x00007ff9de950000.
C:\Program Files\FakeDLLName.dll is loaded at 0x00007ff9de4d0000.
C:\WINDOWS\System32\ADVAPI32.dll is loaded at 0x00007ff9e0780000.
C:\WINDOWS\System32\msvcrt.dll is loaded at 0x00007ff9df9a0000.
C:\WINDOWS\System32\sechost.dll is loaded at 0x00007ff9e0530000.
C:\WINDOWS\System32\RPCRT4.dll is loaded at 0x00007ff9df2d0000.
C:\WINDOWS\System32\bcrypt.dll is loaded at 0x00007ff9decc0000.
C:\WINDOWS\SYSTEM32\FLTLIB.DLL is loaded at 0x00007ff9de460000.
C:\WINDOWS\System32\ucrtbase.dll is loaded at 0x00007ff9defb0000.

[+] Listing hooked modules
-----
Checking ntdll.dll at KERNEL32.DLL IAT
|-- KERNEL32.DLL IAT to ntdll.dll of function NtEnumerateKey is hooked to
0x00007ff9e132d610
|-- KERNEL32.DLL IAT to ntdll.dll of function *NtTerminateProcess is hooked to
0x00007ff9e132d550
|-- KERNEL32.DLL IAT to ntdll.dll of function NtMapUserPhysicalPagesScatter is hooked to
0x00007ff9e132d030
|-- KERNEL32.DLL IAT to ntdll.dll of function NtDeleteValueKey is hooked to
0x00007ff9e132eaa0
|-- KERNEL32.DLL IAT to ntdll.dll of function NtSetValueKey is hooked to
0x00007ff9e132dbc0
...
+-- 81 hooked functions.
```

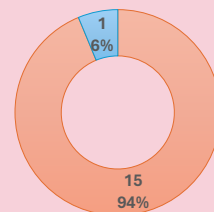
3.5. Resultado

Em virtude da técnica HookChain se tratar de uma técnica executada 100% em modo de usuário (ring 3) e focando evasão neste mesmo nível de privilégio, nenhuma verificação no tocante a existência de validações, hooks e agentes em espaço de Kernel (ring 0) foi realizada. Desta forma a ausência de hooks em ring 3 não implica de forma direta que o HookChain será capaz de evasão total do EDR uma vez que as telemetrias e monitoramentos em ring 0 continuarão ativas.

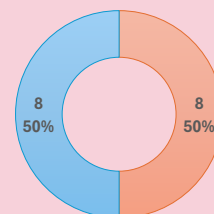
A tabela abaixo apresenta o resultado da enumeração realizada entre os dias 01 e 22 de março de 2024.

PRODUTO	PONTO DE INTERCEPTAÇÃO (HOOK)	
	NTDLL	KERNELBASE / KERNEL32
BitDefender	✓	⊖
CarbonBlack	✓	⊖
Checkpoint	✓	⊖
Cortex	⊖	⊖
CrowdStrike Falcon	✓	⊖
Windows Defender	⊖	⊖
Windows Defender + ATP	⊖	⊖
Elastic	⊖	⊖
ESET	⊖	⊖
Kaspersky	⊖	⊖
MalwareBytes	⊖	⊖
SentinelOne	✓	✓
Sophos	✓	⊖
Symantec	⊖	⊖
Trellix	✓	⊖
Trend	✓	⊖

Resultado 1: 94% das soluções de EDR analisadas (15 de 16) não apresentam hook na camada dos subsistemas acima da Ntdll.dll, ou seja, na verificação de todas DLLs carregadas na aplicação que referenciam a Ntdll, somente uma solução de EDR apresentou hook na IAT.



Resultado 2: 50% das soluções de EDR analisadas (8 de 16) apresentam uma ausência de hooks em modo de usuário.



Nota: Durante os testes finais foram observados a presença de hooks nas DLLs de subsistemas (kernel32 e kernelbase) porém dentro do código das funções críticas, como por exemplo a CreateProcess, e não utilizando hooks da IAT. Para o objetivo deste estudo estes casos não foram considerados nos resultados acima.

4. HOOKCHAIN – A TÉCNICA

4.1. Visão geral

Vamos iniciar essa sessão apresentando uma visão geral e simplificada da técnica foco deste artigo, nomeada **HookChain**. Posteriormente, iniciaremos o detalhamento técnico do HookChain com a apresentação das estruturas de dados e tabelas utilizadas no item 4.2, seguindo com a metodologia utilizada para o preenchimento dessas tabelas em 4.3. Em sequência detalhamos o processo de Hook da IAT no item 4.4, e por fim em 4.5, demonstraremos os testes funcionais e a transparência da presença do implante do HookChain na Stack Call.

De forma geral a técnica HookChain está baseada no seguinte fluxo:

1. Utilização de uma das técnicas de mapeamento dinâmico do SSN apresentadas anteriormente, como o Halo's gate.
2. Mapeamento de algumas funções base para utilização nas ações dos próximos passos, como:
 - a. NtAllocateReserveObject
 - b. NtAllocateVirtualMemory
 - c. NtQueryInformationProcess
 - d. NtProtectVirtualMemory
 - e. NtReadVirtualMemory
 - f. NtWriteVirtualMemory
3. Criação e preenchimento de uma array onde cada item contém o seguinte conteúdo:
 - a. SSN (Syscall Number)
 - b. Endereço da função na Ntdll.dll
 - c. Endereço de memória da instrução SYSCALL mais próxima da função na Ntdll.dll.
4. Realiza o carregamento prévio de outras DLLs, caso seja de conhecimento que a aplicação em execução irá carregar dinamicamente e utilizar outra DLL que ainda não foi carregada no processo atual, bem como a verificação de que se essa DLL a ser carregada realiza chamadas para funções da Ntdll.dll.
5. Utilização das chamada indireta (Indirect Syscall) com as funções mapeadas no item 2 para realizar a leitura, enumeração e tratamento das estruturas das tabelas de exportação e importação de todas as DLLs carregadas.
6. Alteração da IAT das principais DLLs que utilizam as chamadas para

a Ntdll.dll como kernel32, kernelbase, bcrypt, bcryptPrimitives, gdi32, mswsock, netutils e urlmon. Esta ação visa alterar na IAT o endereço de destino das chamadas nativas Nt/Zw para funções internas da nossa aplicação. Desta forma quando uma DLL de subsistema, como a kernel32, for chamar uma função da Ntdll.dll, na realidade será executado o código do implante do HookChain. Concretizando, desta forma, o IAT Hook conforme visto anteriormente no item 2.3.2 deste artigo.

Após a realização dessas ações a utilização das APIs e subsistemas seguem de forma convencional, pois a camada de desvio do fluxo e evasão já foi implantada, não necessitando qualquer ação adicional. De forma que as execuções das chamadas da Ntdll.dll serão realizadas através das funções internas da nossa aplicação, porém de forma transparente para o PE em execução, pois o mesmo continuará utilizando as APIs das subsistemas conforme demonstrados na Figura 14.

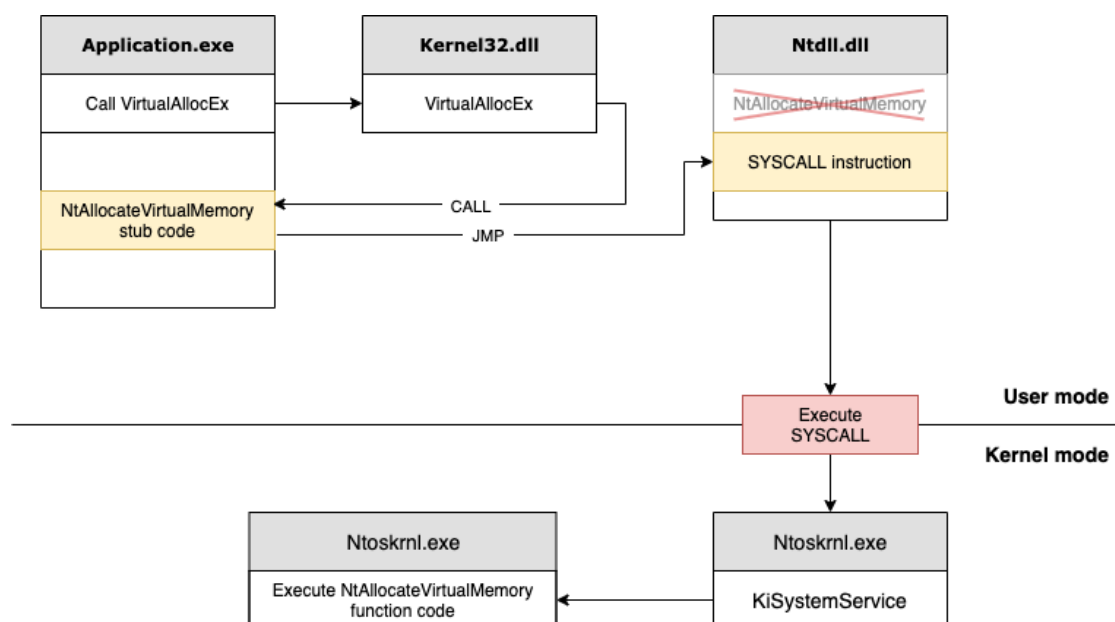


Figura 14: HookChain workflow

Esta metodologia tem a capacidade de evadir todos os hooks em modo do usuário realizados na Ntdll.dll pois todo o controle da execução está na própria aplicação. Tendo as seguintes vantagens perante as outras técnicas aqui apresentadas:

- Redução na probabilidade de identificação pelo EDR em virtude de seguir as regras propostas por algumas telemetrias como:
 - Tempo de execução total do processo, pois o tempo de execução das chamadas se manterá muito próximo ao processo original.
 - Cadeia de execução, onde o EDR espera que a chamada da função tenha vindo da aplicação, posteriormente passado pela

Kernel32.dll, posteriormente pela Ntdll.dll. Isso se dá pelo fato do implante do HookChain (função de interceptação) passar de forma transparente no stack call (conforme veremos em mais detalhes posteriormente).

- Portabilidade
 - Nenhum esforço e/ou alteração necessário para a execução de códigos/aplicações pré-existentes pois a interceptação ocorre de forma ampla e transparente para a aplicação em execução.

4.2. Estruturas de dados e tabelas

4.2.1. Struct SYSCALL_INFO

Conforme visto anteriormente um dos primeiros passos é a criação de uma array com o registro de diversas informações que serão utilizadas no decorrer da execução, sendo assim essa array utiliza como item uma estrutura chamada SYSCALL_INFO conforme abaixo:

```
typedef struct _SYSCALL_INFO {  
    DWORD64 dwSsn;  
    PVOID    pAddress;  
    PVOID    pSyscallRet;  
    PVOID    pStubFunction;  
    DWORD64 dwHash;  
} SYSCALL_INFO, * PSYSCALL_INFO;
```

Onde:

- **dwSsn**: Campo de armazenamento do número do Syscall (SSN).
- **pAddress**: Campo de armazenamento do endereço virtual (Virtual Address) da função dentro da Ntdll.
- **pSyscallRet**: Campo de armazenamento do endereço virtual de uma instrução SYSCALL dentro da Ntdll.
- **pStubFunction**: Campo de armazenamento do endereço da função de interceptação (implante) do HookChain, este é o endereço para o qual serão direcionadas todas as chamadas da função em questão. Em outras palavras este é o endereço que será atribuído na IAT em substituição ao endereço virtual da função da Ntdll.
- **dwHash**: Hash de identificação da função. Este hash é calculado através do nome da função da ntdll. Não é armazenado e utilizado o nome da função para dificultar a identificação por parte dos EDRs.

4.2.2. Struct SYSCALL_LIST

A estrutura SYSCALL_LIST, conforme visualizado abaixo, detém um campo que armazena a quantidade de registros atuais da tabela, e na

sequencia detém uma array com 512 posições com registros do tipo SYSCALL_INFO.

```
#define MAX_ENTRIES 512

typedef struct _SYSCALL_LIST
{
    DWORD64 Count;
    SYSCALL_INFO Entries[MAX_ENTRIES];
} SYSCALL_LIST, * PSYSCALL_LIST;
```

4.2.3. Referencias e índices

A próxima estrutura de dados é na verdade um apontamento da seção .data da nossa aplicação definida em Assembly conforme abaixo:

```
.data
    qTableAddr QWORD 0h
    qListEntrySize QWORD 28h
    qStubEntrySize QWORD 14h

    qIdx0 QWORD 0h
    qIdx1 QWORD 0h
    qIdx2 QWORD 0h
    qIdx3 QWORD 0h
    qIdx4 QWORD 0h
    qIdx5 QWORD 0h
```

Onde:

- **qTableAddr:** Variável onde é armazenado o endereço virtual da instancia da tabela/struct SYSCALL_LIST.
- **qListEntrySize:** Variável que contém o tamanho (em bytes) de cada entrada da lista SYSCALL_LIST->Entries.
- **qStubEntrySize:** Variável que contém o tamanho (em bytes) de cada função de interceptação utilizada pelo HookChain. Maiores detalhes sobre essas funções e metodologia de utilização serão realizados mais a frente neste artigo.
- **qIdx0 - qIdx5:** Variáveis onde serão armazenados em que posição da array estão as informações das funções nativas necessárias para os processos e manipulações iniciais. Essas variáveis cujo os nomes finalizam com os valores de 0 a 5 armazenam o índice das seguintes funções 0 – ZwOpenProcess, 1 – ZwProtectVirtualMemory, 2 – ZwReadVirtualMemory, 3 – ZwWriteVirtualMemory, 4 – ZwAllocateVirtualMemory, 5 – ZwDelayExecution.

4.3. Preenchimento das tabelas de dados

A estrutura de dados `SYSCALL_LIST`, em nosso código, foi definida em uma variável estática de nome `SyscallList` conforme abaixo:

```
static SYSCALL_LIST SyscallList;
```

O preenchimento da array do campo `SyscallList.Entries` é realizado seguindo os seguintes passos:

1. Localiza o endereço base da `Ntdll.dll` utilizando as tabelas TEB (Thread Environment Block) e PEB (Process Environment Block).
2. Enumera todas as funções com o nome iniciado em "Zw" ou "Nt".
3. Verifica se a função em questão é uma das funções que será utilizada incondicionalmente através do Indirect Syscall. Se sim adiciona uma nova entrada na array `SyscallList.Entries` e salva em que posição da array esta função está presente nas variáveis `qIdx0` - `qIdx5`. Segue a lista das funções:
 - a. `NtAllocateReserveObject`
 - b. `NtAllocateVirtualMemory`
 - c. `NtQueryInformationProcess`
 - d. `NtProtectVirtualMemory`
 - e. `NtReadVirtualMemory`
 - f. `NtWriteVirtualMemory`
4. Verifica se a função em questão está com JMP presente em seu código, evidenciando a presença de um hook aplicado pelo EDR. Se sim adiciona uma nova entrada na array `SyscallList.Entries`.

Desta forma ao final deste processo tem-se o array preenchido com todas as funções Nt/Zw que apresentam um hook do EDR, bem como as 6 funções adicionadas incondicionalmente para utilização futura conforme pode-se observar na Figura 15.

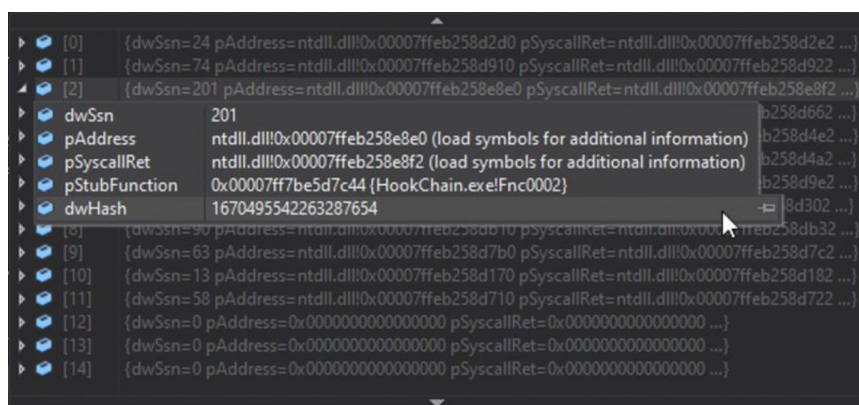


Figura 15: Valores da array `SyscallList.Entries`

```

0:004> uf ntdll!NtCreateUserProcess
ntdll!NtCreateUserProcess:
00007ffe`b258e8e0 4c8bd1          mov     r10,rcx
00007ffe`b258e8e3 b8c9000000       mov     eax,0C9h
00007ffe`b258e8e8 f604250803fe7f01 test     byte ptr [SharedUserData+0x308
(00000000`7ffe0308)],1
00007ffe`b258e8f0 7503             jne     ntdll!NtCreateUserProcess+0x15 (00007ffe`b258e8f5)
Branch

ntdll!NtCreateUserProcess+0x12:
00007ffe`b258e8f2 0f05             syscall
00007ffe`b258e8f4 c3              ret

```

Como se pode ver na imagem e no resultado do comando no Windbg acima o algoritmo do HookChain foi capaz de obter o SSN da função NtCreateUserProcess (decimal 201, hexadecimal 0x00C9), bem como calcular o endereço da próxima instrução SYSCALL (00007ffe`b258e8f2).

Nos passos 3 e 4, o SSN é obtido através do algoritmo de checagem utilizado na técnica Halo's Gate. Sendo implementado conforme o trecho de código abaixo:

```

static DWORD64 GetSSN(_In_ PVOID pAddress)
{
    BYTE low, high;

    /*
        Handle non-hooked functions

        mov r10, rcx
        mov rax, <ssn>
    */
    if (*(PBYTE)pAddress + 0 == 0x4c && *(PBYTE)pAddress + 1 == 0x8b && *(PBYTE)pAddress + 2 == 0xd1 &&
        *(PBYTE)pAddress + 3 == 0xb8 && *(PBYTE)pAddress + 6 == 0x00 && *(PBYTE)pAddress + 7 == 0x00) {

        high = *(PBYTE)pAddress + 5;
        low = *(PBYTE)pAddress + 4;

        return (high << 8) | low;
    }

    // Derive SSN from neighbour syscalls
    for (WORD idx = 1; idx <= MAX_NEIGHBOURS; idx++) {
        if (*(PBYTE)pAddress + 0 + idx * NEXT == 0x4c && *(PBYTE)pAddress + 1 + idx * NEXT == 0x8b &&
            *(PBYTE)pAddress + 2 + idx * NEXT == 0xd1 && *(PBYTE)pAddress + 3 + idx * NEXT == 0xb8 &&
            *(PBYTE)pAddress + 6 + idx * NEXT == 0x00 && *(PBYTE)pAddress + 7 + idx * NEXT == 0x00) {

            high = *(PBYTE)pAddress + 5 + idx * NEXT;
            low = *(PBYTE)pAddress + 4 + idx * NEXT;

            return (high << 8) | low - idx;
        }
    }
}

```

```

        if (((PBYTE)pAddress + 0 + idx * PREV) == 0x4c && (((PBYTE)pAddress + 1 + idx *
PREV) == 0x8b &&
            *((PBYTE)pAddress + 2 + idx * PREV) == 0xd1 && *((PBYTE)pAddress + 3 + idx *
PREV) == 0xb8 &&
            *((PBYTE)pAddress + 6 + idx * PREV) == 0x00 && *((PBYTE)pAddress + 7 + idx *
PREV) == 0x00) {

            high = *((PBYTE)pAddress + 5 + idx * PREV);
            low = *((PBYTE)pAddress + 4 + idx * PREV);

            return (high << 8) | low + idx;

        }
    }

    return -1;
}

```

Já o endereço da próxima instrução SYSCALL é obtido com o seguinte código:

```

static PVOID GetNextSyscallInstruction(_In_ PVOID pStartAddr) {
    for (DWORD i = 0, j = 1; i <= 512; i++, j++) {
        if (((PBYTE)pStartAddr + i) == 0x0f && (((PBYTE)pStartAddr + j) == 0x05) {
            return (PVOID)((ULONG_PTR)pStartAddr + i);
        }
    }
    return NULL;
}

```

Onde é passado como parâmetro o endereço da função na Ntdll, que para exemplo abaixo seria 0x00007ffeb258d0d0, e a função GetNextSyscallInstruction, iniciará a busca neste endereço até localizar a sequência 0x0f05 que representa a instrução SYSCALL.

```

0:004> u ntdll!NtWriteFile
ntdll!NtWriteFile:
00007ffe`b258d0d0 4c8bd1      mov     r10,rcx
00007ffe`b258d0d3 b808000000  mov     eax,8
00007ffe`b258d0d8 f604250803fe7f01  test     byte ptr [SharedUserData+0x308
(00000000`7ffe0308)],1
00007ffe`b258d0e0 7503        jne     ntdll!NtWriteFile+0x15 (00007ffe`b258d0e5)
00007ffe`b258d0e2 0f05        syscall
00007ffe`b258d0e4 c3          ret
00007ffe`b258d0e5 cd2e        int     2Eh
00007ffe`b258d0e7 c3          ret

```

4.4. IAT Hook

Uma vez finalizado o passo anterior, e tendo a array preenchida com os dados das funções nativas Nt/Zw, é possível partir para a próxima fase, que é a fase de alteração da IAT de todas as DLLs carregadas.

Porém se realizarmos o procedimento neste momento e posteriormente

for carregado uma outra biblioteca dinamicamente e essa nova biblioteca contiver em sua IAT a referência para a Ntdll, teríamos que executar novamente o processo de manipulação da IAT dessa DLL. Para evitar este reprocessamento, recomenda-se antes de executar o IAT hook, realizar o carregamento das bibliotecas necessárias.

4.4.1. Carregamento prévio de DLLs

Por exemplo, caso estejamos criando um artefacto utilizando o HookChain e após o implante do HookChain realizarmos a injeção e execução de um Portable Executable (PE) conforme a técnica criada pelo Stephen Fewer, ReflectiveDLLInjection [14], necessitamos realizar a IAT Hook para essas novas DLLs que por ventura foram carregadas pelo ReflectiveDLLInjection. Para evitarmos este processo recomenda-se o mapeamento de quais DLLs o PE utiliza como referência, e quais destas faz chamada direta para a Ntdll e realizar o carregamento e IAT Hook dessas DLLs previamente.

Segue abaixo o trecho de código responsável por realizar o preenchimento da array e IAT Hook das DLLs kernel32 e kernelbase.

```
BOOL UnhookAll(_In_ HANDLE hProcess, _In_ LPCSTR imageName, _In_ BOOLEAN force);

BOOL InitApi(VOID)
{
    if (!FillSyscallTable()) return FALSE;

    UnhookAll((HANDLE)-1, "kernel32", FALSE);
    UnhookAll((HANDLE)-1, "kernelbase", FALSE);

    return TRUE;
}
```

Neste cenário de carregamento prévio que estamos elucidando aqui, bastaria adicionar as DLLs desejadas conforme o exemplo abaixo:

```
BOOL UnhookAll(_In_ HANDLE hProcess, _In_ LPCSTR imageName, _In_ BOOLEAN force);

BOOL InitApi(VOID)
{
    if (!FillSyscallTable()) return FALSE;

    UnhookAll((HANDLE)-1, "kernel32", FALSE);
    UnhookAll((HANDLE)-1, "kernelbase", FALSE);

    UnhookAll((HANDLE)-1, "bcryptPrimitives", TRUE);
    UnhookAll((HANDLE)-1, "ws2_32", TRUE);

    return TRUE;
}
```

4.4.2. IAT Hook

O procedimento de hook da IAT segue da mesma forma que detalhamos no item 2.3.2 deste artigo. De forma geral, o HookChain irá realizar o seguinte procedimento para as DLLs requisitadas através da função `UnhookAll` (demonstrada acima).

1. Listagem (na IAT) de todas as dependências da DLL.
2. Verificação das referências para a `Ntdll`.
3. Verificação de a função referenciada está na array `SyscallList.Entries`, se sim altera o endereço da IAT para o endereço de uma função de interceptação criada pelo HookChain cujo nome está diretamente relacionado ao índice do item na array `SyscallList.Entries`.

4.4.3. Fluxo de execução

Após a finalização dos passos anteriores todos os procedimentos necessários para o implante do HookChain estão finalizados, de forma que a partir deste momento todas as chamadas realizadas para os subsistemas do Windows já estarão livres das interceptações e monitoramentos do EDR a nível da `Ntdll.dll`.

Desta forma, vamos entender mais a fundo o fluxo de execução da aplicação após a finalização dos implantes do HookChain.

Nota: A escolha da função `CreateProcess` para o exemplo e análise a seguir se trata de apenas uma questão didática motivada pelo amplo conhecimento da função em questão, não tendo assim a função `CreateProcess` qualquer relação com o funcionamento do HookChain. Ressaltando, desta forma, que o HookChain não necessita da criação de um novo processo para funcionar pois todos os implantes são realizados no contexto do próprio processo.

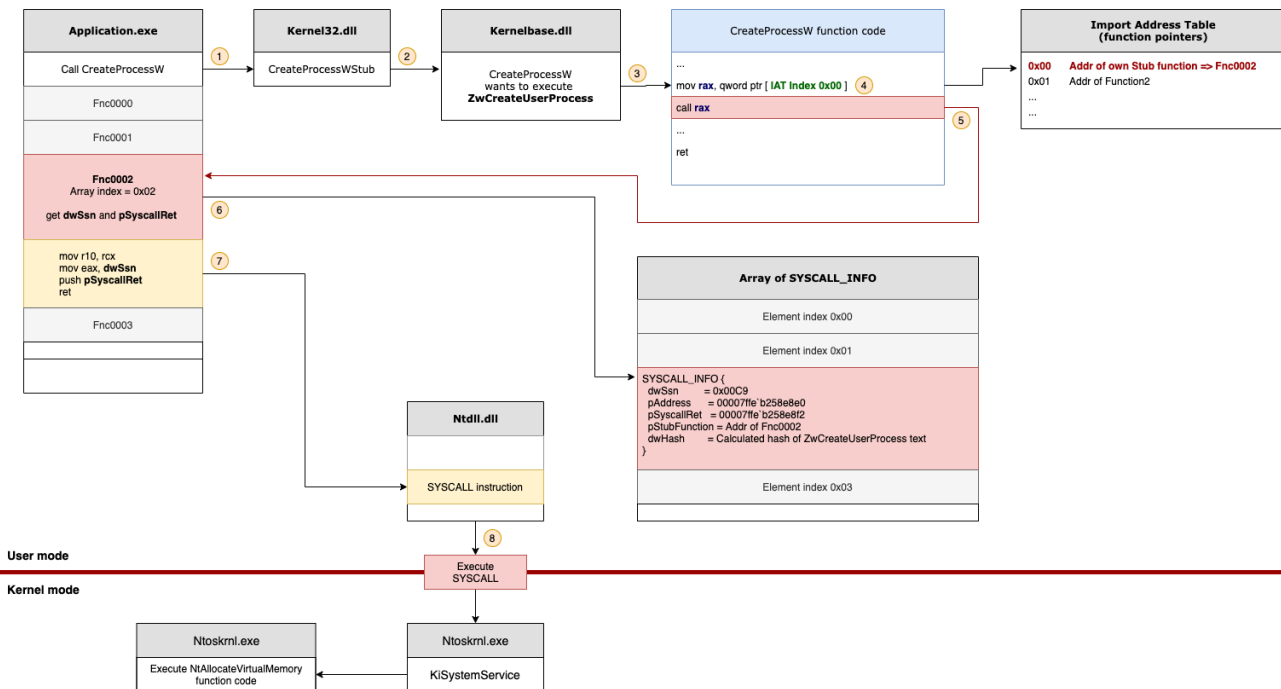


Figura 16: Fluxo de execução após implante HookChain

A Figura 16 demonstra de forma detalhada o fluxo de execução de uma chamada de função após o implante do HookChain, conforme segue o descritivo abaixo:

1. Como exemplificação a aplicação deseja realizar a criação de um novo processo através da função `CreateProcessW` disponível na API/subsistema `Kernel32.dll`.
2. Como essa função específica está implantada na `Kernelbase.dll`, a `kernel32.dll` apenas redireciona o fluxo de execução para a `kernelbase`.
3. Dentro do código da `CreateProcessW` na DLL `kernelbase`, após algumas verificações de parâmetros chegará o ponto de executar a função `ZwCreateUserProcess` pertencente a `Ntdll.dll`.
4. Desta forma o código da `CreateProcessW` irá buscar na IAT da `kernelbase`, onde originalmente teria o endereço da função `ZwCreateUserProcess` na `Ntdll.dll`, porém após o implante da `HookChain`, nessa posição da IAT conterá o endereço da função implantada pelo `HookChain`.
5. Após obter o endereço da função implantada, o código da `CreateProcessW` irá realizar o `CALL` para este endereço ao invés do endereço da `ZwCreateUserProcess` na `Ntdll.dll`, indo desta forma, para a função implantada pelo `HookChain`.
6. Cada função de interceptação da `HookChain` foi criada com um nome/índice específico, em nosso cenário de exemplo o nome da

função é Fnc0002, então o índice correspondente na array SyscallList.Entries será o 0x0002, desta forma o código da HookChain irá buscar na tabela (array SyscallList.Entries[0x0002]) as informações previamente armazenadas como o SSN e o endereço da instrução syscall na Ntdll.

7. De posse de todas as informações necessárias o código da HookChain reproduz o que seria realizado pela função na Ntdll (mov r10, rcx; mov eax, SSN) e posteriormente encaminha o fluxo de execução para o endereço da Ntdll que contém a instrução syscall.
8. Neste ponto do nosso fluxo no topo da pilha conterá o endereço de retorno que será o endereço da próxima instrução inserido na pilha no momento que a CreateProcessW da kernelbase realizou o CALL. Então a Ntdll executa a instrução de syscall. E quando houver o retorno do kernel, o fluxo será encaminhado para o respectivo endereço de retorno dentro da CreateProcessW.

4.5. Testes funcionais e telemetria da stack call

Este teste tem por objetivo a visualização e entendimento da pilha de chamada das funções antes e depois dos implantes do HookChain.

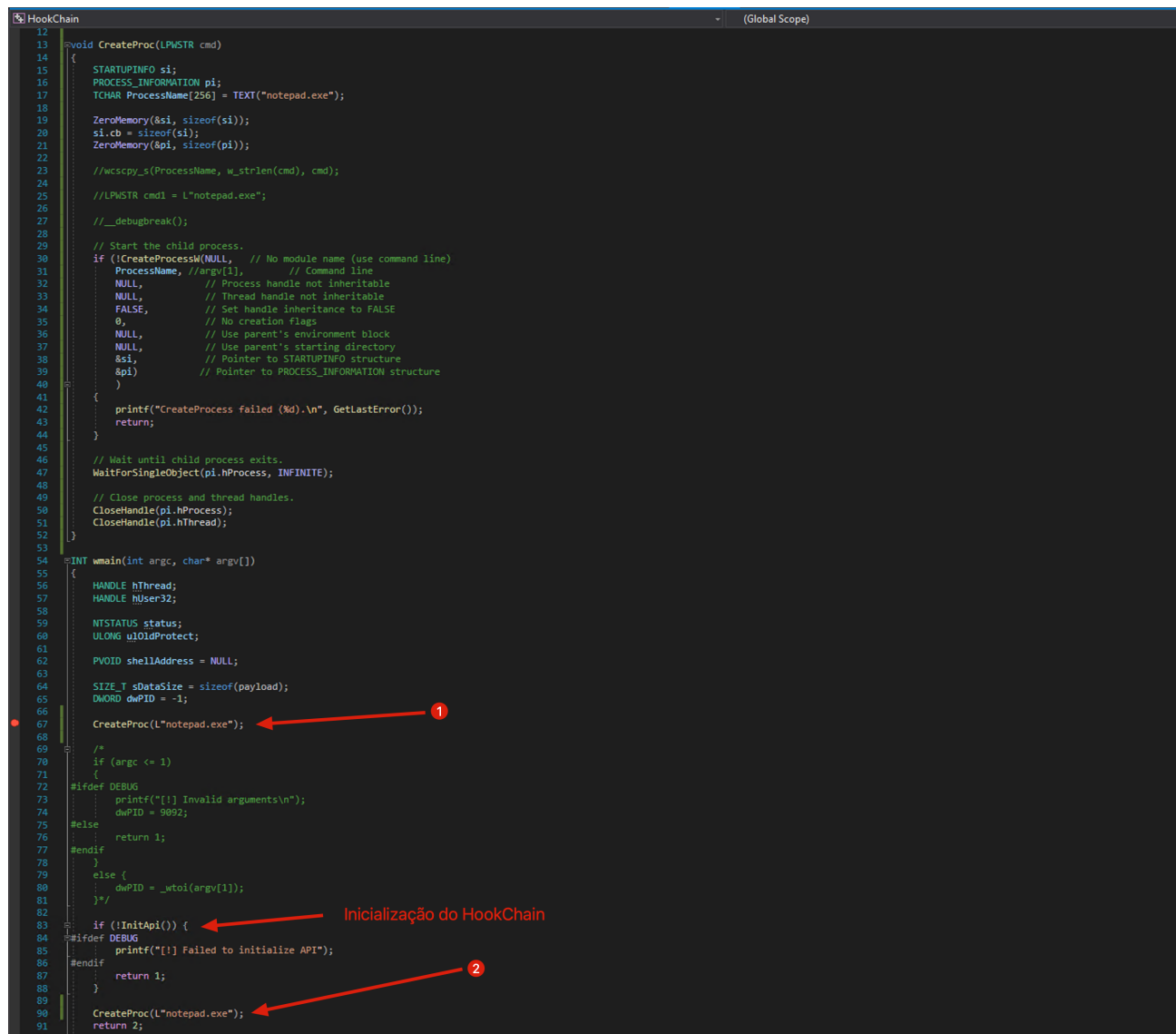
Neste ponto, precisamos ter o cuidado para não confundir o resultado obtido (onde os implantes do HookChain passam despercebidos na pilha de chamada) com outra técnica de bypass conhecida como "Call stack spoofing", ou em uma tradução livre, falsificação da pilha de chamadas, de forma que o Call stack spoofing, se utiliza da manipulação da pilha para inserir arbitrariamente endereços de retorno fazendo com que pareça ter vindo das funções originais. Por outro lado, a técnica HookChain não realiza "Call stack spoofing" nem tampouco a manipulação arbitrária da pilha, pois ela efetivamente intercepta de forma transparente as chamadas para as APIs da NTDll (como vimos anteriormente).

Como vimos anteriormente os implantes do HookChain são genéricos e realizados antes das ações desejadas. Para o exemplo abaixo analisaremos toda a trilha de chamadas da função CreateProcessW.

Sendo assim o código do teste realiza 3 ações:

1. Inicia o processo Notepad.exe com a utilização da API CreateProcessW. Este procedimento é realizado antes dos implantes do HookChain.
2. Todos os implantes do HookChain são realizados.

3. Inicia um novo processo Notepad.exe com a utilização da API CreateProcessW. Como neste momento o HookChain já realizou todos os seus implantes e bypasses, a função original ZwCreateUserProcess da Ntdll.dll não será executada.



```
12 void CreateProc(LPWSTR cmd)
13 {
14     STARTUPINFO si;
15     PROCESS_INFORMATION pi;
16     TCHAR ProcessName[256] = TEXT("notepad.exe");
17
18     ZeroMemory(&si, sizeof(si));
19     si.cb = sizeof(si);
20     ZeroMemory(&pi, sizeof(pi));
21
22     //wscpy_s(ProcessName, w_strlen(cmd), cmd);
23
24     //LPWSTR cmd1 = L"notepad.exe";
25
26     //__debugbreak();
27
28     // Start the child process.
29     if (!CreateProcessW(NULL, // No module name (use command line)
30         ProcessName, //argv[1], // Command line
31         NULL, // Process handle not inheritable
32         NULL, // Thread handle not inheritable
33         FALSE, // Set handle inheritance to FALSE
34         0, // No creation flags
35         NULL, // Use parent's environment block
36         NULL, // Use parent's starting directory
37         &si, // Pointer to STARTUPINFO structure
38         &pi) // Pointer to PROCESS_INFORMATION structure
39     )
40     {
41         printf("CreateProcess failed (%d).\n", GetLastError());
42         return;
43     }
44
45     // Wait until child process exits.
46     WaitForSingleObject(pi.hProcess, INFINITE);
47
48     // Close process and thread handles.
49     CloseHandle(pi.hProcess);
50     CloseHandle(pi.hThread);
51 }
52
53
54 INT wmain(int argc, char* argv[])
55 {
56     HANDLE hThread;
57     HANDLE hUser32;
58
59     NTSTATUS status;
60     ULONG ulOldProtect;
61
62     PVOID shellAddress = NULL;
63
64     SIZE_T sDataSize = sizeof(payload);
65     DWORD dwPID = -1;
66
67     CreateProc(L"notepad.exe"); // 1
68
69     /*
70     if (argc <= 1)
71     {
72         #ifdef DEBUG
73             printf("[!] Invalid arguments\n");
74             dwPID = 9002;
75         #else
76             return 1;
77         #endif
78     }
79     else {
80         dwPID = _wtoi(argv[1]);
81     }*/
82
83     if (!InitApi()) { // Inicialização do HookChain
84         #ifdef DEBUG
85             printf("[!] Failed to initialize API");
86         #endif
87         return 1;
88     }
89
90     CreateProc(L"notepad.exe"); // 2
91     return 2;
92 }
```

Figura 17: Código utilizado para este teste

Process Monitor - Sysinternals: www.sysinternals.com							
File Edit Event Filter Tools Options Help							
Time of Day	Process Name	PID	Operation	Path	Result	Detail	
4:08:10.4365365 PM	HookChain.exe	12892	c# Process Start		SUCCESS	Parent PID: 1760, ...	
4:08:10.4365420 PM	HookChain.exe	12892	c# Thread Create		SUCCESS	Thread ID: 7176	
4:08:10.4397076 PM	HookChain.exe	12892	c# Load Image	E:\Projects\Personal\HookChain\HookChain.exe	SUCCESS	Image Base: 0x7f...	
4:08:10.4398129 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\ntdll.dll	SUCCESS	Image Base: 0x7f...	
4:08:10.4406560 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\kernel32.dll	SUCCESS	Image Base: 0x7f...	
4:08:10.4408618 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\KernelBase.dll	SUCCESS	Image Base: 0x7f...	
4:08:10.4438025 PM	HookChain.exe	12892	c# Process Create	C:\WINDOWS\System32\Conhost.exe	SUCCESS	PID: 13168, Comm...	
4:08:10.5187230 PM	HookChain.exe	12892	c# Thread Create		SUCCESS	Thread ID: 8036	
4:08:10.5192031 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\vcruntime140d.dll	SUCCESS	Image Base: 0x7f...	
4:08:10.5196666 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\ucrtbased.dll	SUCCESS	Image Base: 0x7f...	
4:08:10.5198366 PM	HookChain.exe	12892	c# Thread Create		SUCCESS	Thread ID: 12360	
4:08:10.5209022 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\sechost.dll	SUCCESS	Image Base: 0x7f...	
4:08:10.5209774 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\iprct4.dll	SUCCESS	Image Base: 0x7f...	
4:08:10.5210645 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\bcrypt.dll	SUCCESS	Image Base: 0x7f...	
4:08:10.5254013 PM	HookChain.exe	12892	c# Process Create	C:\WINDOWS\SYSTEM32\notepad.exe	SUCCESS	PID: 13408, Comm...	
4:08:29.2847231 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\bcryptprimitives.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2845903 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\imm32.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2850703 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\user32.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2851448 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\win32u.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2852191 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\gdi32.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2853033 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\gdi32full.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2853801 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\mmdevp_wm.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2854607 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\ucrtbase.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2855423 PM	HookChain.exe	12892	c# Thread Create		SUCCESS	Thread ID: 6660	
4:08:29.2878706 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\clbcatq.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2879564 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\msvcp.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2880406 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\combase.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2891467 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\verutil.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2893489 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\advapi32.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2893535 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\SHCore.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2896655 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\rsaui.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2901509 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\IPHLPAPI.DLL	SUCCESS	Image Base: 0x7f...	
4:08:29.2905434 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\winhttp.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2909390 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\urlmon.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2916122 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\advapi32.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2916341 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\ntutils.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2922407 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\newsoc.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2901509 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\IPHLPAPI.DLL	SUCCESS	Image Base: 0x7f...	
4:08:29.2927689 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\winhttp.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2936020 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\cryptapi.dll	SUCCESS	Image Base: 0x7f...	
4:08:29.2939323 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\rsaenh.dll	SUCCESS	Image Base: 0x7f...	
4:08:34.3432663 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\ntutils.dll	SUCCESS	Image Base: 0x7f...	
4:08:34.3595639 PM	HookChain.exe	12892	c# Process Create	C:\WINDOWS\SYSTEM32\notepad.exe	SUCCESS	PID: 12688, Comm...	
4:09:10.5333906 PM	HookChain.exe	12892	c# Thread Exit		SUCCESS	Thread ID: 12360, ...	
4:09:10.5334268 PM	HookChain.exe	12892	c# Thread Exit		SUCCESS	Thread ID: 8036, ...	
4:09:10.5334589 PM	HookChain.exe	12892	c# Thread Exit		SUCCESS	Thread ID: 6660, ...	
4:09:10.5334589 PM	HookChain.exe	12892	c# Thread Exit		SUCCESS	Thread ID: 6660, ...	
4:08:34.3432663 PM	HookChain.exe	12892	c# Load Image	C:\Windows\System32\winhttp.dll	SUCCESS	Image Base: 0x7f...	
4:08:34.3595639 PM	HookChain.exe	12892	c# Process Create	C:\WINDOWS\SYSTEM32\notepad.exe	SUCCESS	PID: 12688, Comm...	
4:09:10.5333906 PM	HookChain.exe	12892	c# Thread Exit		SUCCESS	Thread ID: 12360, ...	

Figura 18: Monitoramento da execução

Event Properties					
Event Process Stack					
Frame	Module	Location	Address	Path	
K 0	ntoskml.exe	PspCallProcessNotifyRoutines + 0x213	0xffff8060ea6455f	C:\WINDOWS\system32\ntoskml.exe	
K 1	ntoskml.exe	PspInsertThread + 0x68e	0xffff8060ea8c9ea	C:\WINDOWS\system32\ntoskml.exe	
K 2	ntoskml.exe	NtCreateUserProcess + 0xd0e	0xffff8060e9edda0	C:\WINDOWS\system32\ntoskml.exe	
K 3	ntoskml.exe	KiSystemServiceCopyEnd + 0x25	0xffff8060e811575	C:\WINDOWS\system32\ntoskml.exe	
U 4	ntdll.dll	ZwCreateUserProcess + 0x14	0x7feb258e9f4	C:\Windows\System32\ntdll.dll	
U 5	KernelBase.dll	CreateProcessInternalW + 0x22e2	0x7feafc2c5f2	C:\Windows\System32\KernelBase.dll	
U 6	KernelBase.dll	CreateProcessW + 0x66	0x7feafc29666	C:\Windows\System32\KernelBase.dll	
U 7	kernel32.dll	CreateProcessWStub + 0x54	0x7feb1b8cec4	C:\Windows\System32\kernel32.dll	
U 8	HookChain.exe	CreateProc + 0xf0, E:\Projects\Personal\HookChain\HookChain\main.c(21)	0x7f7be5d62f0	E:\Projects\Personal\HookChain\HookChain\main.c(21)	
U 9	HookChain.exe	wmain + 0x78, E:\Projects\Personal\HookChain\HookChain\main.c(67)	0x7f7be5d6438	E:\Projects\Personal\HookChain\HookChain\main.c(67)	
U 10	HookChain.exe	invoke_main + 0x39, D:\a_work\1\s\src\vc\tools\crt\vcstartup\src\startup\exe_common.inl(90)	0x7f7be5db7b9	E:\Projects\Personal\HookChain\HookChain\main.c(67)	
U 11	HookChain.exe	__scrt_common_main_seh + 0x12e, D:\a_work\1\s\src\vc\tools\crt\vcstartup\src\startup\exe_common.inl(281)	0x7f7be5db65e	E:\Projects\Personal\HookChain\HookChain\main.c(67)	
U 12	HookChain.exe	__scrt_common_main + 0xe, D:\a_work\1\s\src\vc\tools\crt\vcstartup\src\startup\exe_common.inl(330)	0x7f7be5db51e	E:\Projects\Personal\HookChain\HookChain\main.c(67)	
U 13	HookChain.exe	wmainCRTStartup + 0xe, D:\a_work\1\s\src\vc\tools\crt\vcstartup\src\startup\exe_common.inl(16)	0x7f7be5db84e	E:\Projects\Personal\HookChain\HookChain\main.c(67)	
U 14	kernel32.dll	BaseThreadInitThunk + 0x14	0x7feb1b87344	C:\Windows\System32\kernel32.dll	
U 15	ntdll.dll	RtlUserThreadStart + 0x21	0x7feb25426b1	C:\Windows\System32\ntdll.dll	

Figura 19: Stack trace da chamada CreateProcessW antes dos implantes

```

E:\Projects\Personal\HookChain\HookChain\HookChain.exe
0x00007FF7BE63DD30 = &SyscallList
0x000001CF58190000 = pImageExportDirectory
0x000001CF58190000 = lpLocalAddress
e[0] ZwAllocateVirtualMemory 24 0x00007FFEB258D2D0 0x989246E5A13FCBD9
e[1] ZwCreateSection 74 0x00007FFEB258D910 0x71FE19F80D84EC5D
e[2] ZwCreateUserProcess 201 0x00007FFEB258E8E0 0x172ECAD8537A0F66
e[3] ZwDelayExecution 52 0x00007FFEB258D650 0x8599A0E7F8A94577
e[4] ZwMapViewOfSection 40 0x00007FFEB258D4D0 0x09C2657B2A8355D7
e[5] ZwOpenProcess 38 0x00007FFEB258D490 0x8AD1C604A65844A5
e[6] ZwProtectVirtualMemory 80 0x00007FFEB258D9D0 0x7AF7191D67000D85
e[7] ZwQueryInformationProcess 25 0x00007FFEB258D2F0 0x855A46C125055C2F
e[8] ZwQuerySystemTime 90 0x00007FFEB258D810 0x66C71BD1B0714D3E
e[9] ZwReadVirtualMemory 63 0x00007FFEB258D7B0 0x85E26B87B62C2CF0
e[10] ZwSetInformationThread 13 0x00007FFEB258D170 0xB19FEB816B2F7D91E
e[11] ZwWriteVirtualMemory 58 0x00007FFEB258D710 0xF4CE15C0758B33F
[+] Mapped 12 functions
[*] Name: kernel32

```

Figura 20: Array SyscallList.Entries populada e implantes realizados

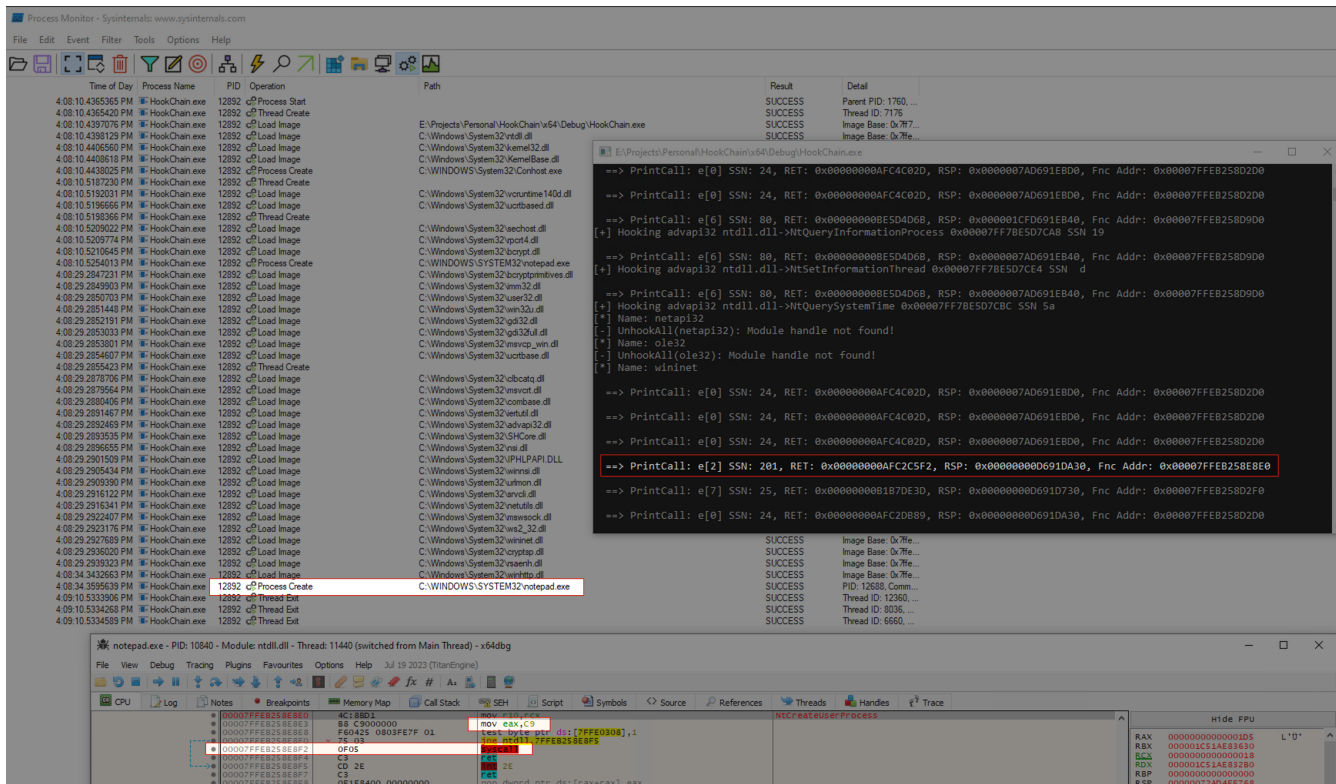


Figura 21: Execução da chamada CreateProcessW após os implantes

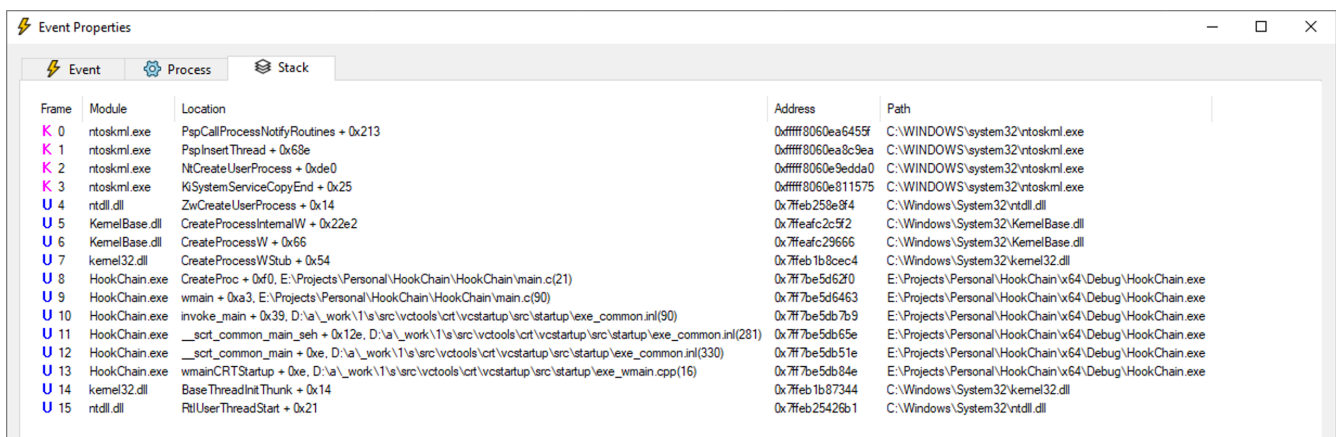


Figura 22: Stack trace da chamada CreateProcessW após os implantes

Pode-se observar na Figura 21 que a aplicação (devido a presença de código de debug) exibiu em tela o momento em que a função de interceptação foi executada bem como o índice na array, SSN e etc.

Na comparação com a Figura 19 e a Figura 22 se pode observar que um dos nossos objetivos foi 100% cumprido, de forma que o desvio do fluxo da aplicação e consequente presença do hook criado pelo HookChain não causou alteração da Stack Trace, podendo desta forma passar despercebido pela telemetria do EDR.

Resultado 3: Telemetria da Stack trace inalterada a ponto do desvio do fluxo (Hook poder passar despercebido por uma verificação de EDR em kernel-land.

```

0x00007FF7BE63DD30 = &SyscallList

Index, Name, Ssn, Ntdll.dll Address
e[0] ZwAllocateVirtualMemory 24 0x00007FFEB258D2D0
e[1] ZwCreateSection 74 0x00007FFEB258D910
e[2] ZwCreateUserProcess 201 0x00007FFEB258E8E0
e[3] ZwDelayExecution 52 0x00007FFEB258D650
e[4] ZwMapViewOfSection 40 0x00007FFEB258D4D0
e[5] ZwOpenProcess 38 0x00007FFEB258D490
e[6] ZwProtectVirtualMemory 80 0x00007FFEB258D9D0
e[7] ZwQueryInformationProcess 25 0x00007FFEB258D2F0
e[8] ZwQuerySystemTime 90 0x00007FFEB258DB10
e[9] ZwReadVirtualMemory 63 0x00007FFEB258D7B0
e[10] ZwSetInformationThread 13 0x00007FFEB258D170
e[11] ZwWriteVirtualMemory 58 0x00007FFEB258D710

```

No texto acima, extraído da console da aplicação no momento da execução, pode-se ver as informações da função ZwCreateUserProcess.

```

0:004> lm
start                end                module name
00007ff7`be5c0000 00007ff7`be64e000 HookChain
00007ffe`48080000 00007ffe`482a1000 ucrtbased
00007ffe`9da80000 00007ffe`9daae000 VCRUNTIME140D
00007ffe`afba0000 00007ffe`afbc7000 bcrypt
00007ffe`afbd0000 00007ffe`afec6000 KERNELBASE
00007ffe`b1680000 00007ffe`b1720000 sechost
00007ffe`b1b70000 00007ffe`b1c2d000 KERNEL32
00007ffe`b2020000 00007ffe`b2145000 RPCRT4
00007ffe`b24f0000 00007ffe`b26e8000 ntdll

0:004> !dh 00007ffe`afbd0000 -f

File Type: DLL
...
2A1560 [ EF64] address [size] of Export Directory
2B04C4 [ 64] address [size] of Import Directory
2CC000 [ 548] address [size] of Resource Directory
2BB000 [ FA38] address [size] of Exception Directory
2EF800 [ 9018] address [size] of Security Directory
2CD000 [ 28B1C] address [size] of Base Relocation Directory
216A10 [ 70] address [size] of Debug Directory
0 [ 0] address [size] of Description Directory
0 [ 0] address [size] of Special Directory
1E3C20 [ 28] address [size] of Thread Storage Directory
193E80 [ 118] address [size] of Load Configuration Directory
0 [ 0] address [size] of Bound Import Directory
1E48B8 [ 1688] address [size] of Import Address Table Directory
29EDD0 [ 4C0] address [size] of Delay Import Directory
0 [ 0] address [size] of COR20 Header Directory
0 [ 0] address [size] of Reserved Directory

```

No trecho acima pode-se observar a listagem dos módulos da aplicação no windbg, bem como o endereço do subsistema Kernelbase e da sua respectiva IAT.

```

0:004> dps 00007ffe`afbd0000 + 1E48B8 00007ffe`afbd0000 + 1E48B8 + 1688
00007ffe`afdb48b8 00007ffe`b2566e70 ntdll!ApiSetQueryApiSetPresence
...
00007ffe`afdb4e20 00007ffe`b258d910 ntdll!NtCreateSection
00007ffe`afdb4e28 00007ffe`b2506790 ntdll!RtlOpenCurrentUser
00007ffe`afdb4e30 00007ffe`b258d4d0 ntdll!NtMapViewOfSection
00007ffe`afdb4e38 00007ffe`b258d270 ntdll!NtQueryDefaultLocale
00007ffe`afdb5038 00007ffe`b258d2f0 ntdll!NtQueryInformationProcess
00007ffe`afdb5040 00007ffe`b2591130 ntdll!RtlCaptureContext
00007ffe`afdb55b8 00007ffe`b258d170 ntdll!NtSetInformationThread
00007ffe`afdb5760 00007ffe`b258d7b0 ntdll!NtReadVirtualMemory
00007ffe`afdb5788 00007ffe`b258d9d0 ntdll!NtProtectVirtualMemory
00007ffe`afdb5790 00007ffe`b258d710 ntdll!NtWriteVirtualMemory
00007ffe`afdb5798 00007ffe`b258d2d0 ntdll!NtAllocateVirtualMemory
00007ffe`afdb57a0 00007ffe`b258de80 ntdll!NtAllocateVirtualMemoryEx
00007ffe`afdb59b0 00007ffe`b258d650 ntdll!NtDelayExecution
00007ffe`afdb5a08 00007ffe`b258d490 ntdll!NtOpenProcess
00007ffe`afdb5cc0 00007ffe`b258e8e0 ntdll!NtCreateUserProcess
...

```

No trecho acima observa-se a IAT da Kernelbase antes dos implantes do HookChain, e no trecho abaixo se pode ver a IAT após os implantes do HookChain, evidenciando desta forma, a alteração efetivada.

```

0:004> dps 00007ffe`afbd0000 + 1E48B8 00007ffe`afbd0000 + 1E48B8 + 1688
00007ffe`afdb48b8 00007ffe`b2566e70 ntdll!ApiSetQueryApiSetPresence
...
00007ffe`afdb4e20 00007ff7`be5d7c30 HookChain!Fnc0001
00007ffe`afdb4e28 00007ffe`b2506790 ntdll!RtlOpenCurrentUser
00007ffe`afdb4e30 00007ff7`be5d7c6c HookChain!Fnc0004
00007ffe`afdb4e38 00007ffe`b258d270 ntdll!NtQueryDefaultLocale
00007ffe`afdb5038 00007ff7`be5d7ca8 HookChain!Fnc0007
00007ffe`afdb5040 00007ffe`b2591130 ntdll!RtlCaptureContext
00007ffe`afdb55b8 00007ff7`be5d7ce4 HookChain!Fnc000A
00007ffe`afdb5760 00007ff7`be5d7cd0 HookChain!Fnc0009
00007ffe`afdb5788 00007ff7`be5d7c94 HookChain!Fnc0006
00007ffe`afdb5790 00007ffe`b258d710 ntdll!NtWriteVirtualMemory
00007ffe`afdb5798 00007ff7`be5d7c1c HookChain!Fnc0000
00007ffe`afdb59b0 00007ff7`be5d7c58 HookChain!Fnc0003
00007ffe`afdb5a08 00007ff7`be5d7c80 HookChain!Fnc0005
00007ffe`afdb5cc0 00007ff7`be5d7c44 HookChain!Fnc0002
...

```

No trecho de código assembly abaixo observa-se as funções para onde as chamadas são encaminhadas. Pode-se observar que cada uma delas tem em seu nome um identificador e em seu código este identificador é utilizado como referência da array SyscallList.Entries para a obtenção da informações previamente preenchidas.

```

Fnc0000 PROC
    mov rax, SyscallExec
    push rax
    mov rax, 0000h
    ret
    nop
Fnc0000 ENDP

```

```

Fnc0001 PROC
    mov rax, SyscallExec
    push rax
    mov rax, 0001h
    ret
    nop
Fnc0001 ENDP

```

```

Fnc0002 PROC
    mov rax, SyscallExec
    push rax
    mov rax, 0002h
    ret
    nop
Fnc0002 ENDP

```

Abaixo tem-se o código assembly da função SyscallExec, função esta responsável por utilizar o indexador das funções que receberão o fluxo da execução interceptada, buscar na array SyscallList.Entries as respectivas informações e direcionar o fluxo da aplicação para o endereço da instrução Syscall dentro da Ntdll.dll.

```

SyscallExec PROC
    sub rsp, 08h    ; Address to place syscall addr and use with ret
    push r12
    push r9
    push r8
    push rdx
    push rcx
    push rbp
    mov rbp, rsp
    mov r12, rdx
    mov rdx, qListEntrySize
    mul rdx
    mov rdx, r12
    mov r12, qTableAddr
    lea rax, [r12 + rax]
    mov r12, [rax + 10h]
    mov rax, [rax]
    mov [rbp + 30h], r12    ; 0x30 = 6 * 8 = 48
    mov rsp, rbp
    pop rbp
    pop rcx
    pop rdx
    pop r8
    pop r9
    pop r12
    mov r10, rcx
    ret    ; jmp to the address saved at stack
SyscallExec ENDP

```

5. HOOKCHAIN – TESTES

5.1. Metodologia de testes

Diferentemente da enumeração prévia descrita no item 3 deste artigo, esta fase de testes foi realizada totalmente por mim em um ambiente controlado.

Para a realização dos testes foram selecionados 14 produtos de EDR, sendo que 9 estão presentes no quadrante mágico do Gartner de 31 de dezembro de 2023 [15].



Figura 23: Quadrante mágico do Gartner para plataformas de proteção de endpoint de 31 de dezembro de 2023 [15].

Os produtos presentes no quadrante mágico do Gartner testados foram apresentados na imagem acima marcados em verde, já os produtos marcados em cinza não puderam ser testados.

Adicionalmente outros 4 produtos, não presentes no quadrante do Gartner foram testados, sendo eles: Acronis, Cylance, Elastic e MalwareBytes

Para estes testes foram preparadas duas versões do HookChain conforme descritas abaixo:

5.1.1. Injeção em processo remoto

Essa primeira versão tem por objetivo realizar a injeção de um shellcode simples em um processo remoto utilizando uma técnica amplamente

conhecida de criação de uma thread em um processo remoto. O Shellcode injetado é criado em tempo de execução para a execução da chamada de API MessageBox da User32.dll.

Este executável segue o seguinte fluxo:

1. Realização dos implantes do HookChain
2. Criação em tempo de execução de um código (em Assembly) para executar o MessageBox
3. Abertura de um handle para o processo onde será injetado o código
4. Criação de uma área de memória no processo remoto
5. Injeção do código assembly no processo remoto
6. Criação e execução de uma thread remota apontando para o assembly carregado

5.1.2. Download e execução de um shellcode

Já para a segunda versão do HookChain foi preparado uma versão responsável por realizar o download de um shellcode via HTTP e executá-lo no próprio processo. Essa estratégia permitiu utilizarmos um único executável do HookChain para diversos payloads diferentes, pois bastou trocar o shellcode no servidor HTTP que o mesmo é injetado e executado em nosso processo.

Este executável segue o seguinte fluxo:

1. Realização dos implantes do HookChain
2. Download via HTTP de um shellcode (código assembly)
3. Injeção do shellcode no processo local
4. Execução do shellcode carregado em memória, em outras palavras de forma refletida.

Pela liberdade fornecida por esta estratégia, diversos payloads, e com diferentes níveis de acesso puderam ser testados, conforme a tabela abaixo:

PAYLOAD	NÍVEL DE ACESSO	OBJETIVO
Metasploit Meterpreter	Usuário comum	Verificar a capacidade de análise e identificação de payload altamente conhecido e sumariamente bloqueado pelos EDRs

Havoc	Usuário comum	Verificar a a capacidade de análise e identificação de payload e beacon de C2 (Command & Control)
Metasploit Meterpreter + Módulo Kiwi	Usuário administrativo	Verificar a a capacidade de análise e identificação de payload altamente conhecido e sumariamente bloqueado pelos EDRs para dump de credenciais
Mimikatz	Usuário administrativo	Verificar a a capacidade de análise e identificação de payload altamente conhecido e sumariamente bloqueado pelos EDRs para dump de credenciais
Procdump	Usuário administrativo	Verificar a a capacidade de identificação de dump de credenciais
LSASS Dump via código proprietário	Usuário administrativo	Verificar a a capacidade de identificação de dump de credenciais

Como se pode observar na tabela acima essa estratégia permite uma variedade quase infinita de possibilidades.

Nota: Vale ressaltar que o objetivo da técnica HookChain é pura e simplesmente o bypass dos pontos de monitoramento das camadas de defesa em nível de usuário (ring 3), sendo assim o mesmo pode ser utilizado em qualquer nível de acesso que o usuário tenha (não administrativo, administrativo ou até como SYSTEM).

Os testes realizados focando a obtenção de credenciais (credential dump) através da memória do LSASS foram realizados com o objetivo de ilustrar as potencialidades da técnica do HookChain quando aliada a outras técnicas novas e/ou conhecidas. Reforçando o fato de o HookChain ser uma técnica de base, possibilitando a execução de outros códigos e/ou técnicas posteriormente aos seus implantes de bypass.

5.2. Resultado

A tabela abaixo apresenta o resultado sumarizados dos testes.

PRODUTO	CÓDIGO EXECUTADO						
	Injeção em processo remoto	Download e execução de um shellcode					
		Meterpreter	Havoc	Meterpreter + Kiwi	Mimikatz	Procdump	LSASS Dump
Acronis	✓	✓	✓	✓	✓	✓	✓
BitDefender	✓	⊖	✓	⊖	⚠	⊖	⚠
Cortex	⊖	⊖	✓	⊖	⊖	⊖	✓
CrowdStrike Falcon	✓	✓	✓	⊖	⚠	⚠	✓
Cylance	⊖	⚠	⊖	⊖	✓	⊖	⊖
Windows Defender	✓	⚠	✓	⊖	⊖	⊖	✓
Windows Defender XDR	✓	✓	✓	⊖	⊖	⊖	✓
Elastic	✓	⊖	⊖	⊖	⊖	⊖	⊖
ESET	✓	⚠	✓	✓	✓	⊖	✓
MalwareBytes	✓	✓	✓	✓	✓	✓	✓
SentinelOne	⚠	✓	⚠	✓	✓	⚠	⚠
Sophos	✓	✓	✓	⚠	⚠	⊖	⚠
Trellix	✓	✓	✓	✓	✓	⊖	✓
Trend	✓	✓	✓	✓	✓	⊖	✓

Legenda:

- ✓ Sucesso na execução sem nenhum alerta ou bloqueio
- ⚠ Execução parcial, execução sem sucesso e sem alerta ou execução com sucesso, porém com alerta
- ⊖ Falha na execução com bloqueio e alertas

5.2.1. Baixo privilégio

Nos testes gerais com usuário de baixo privilégio (Injeção em processo remoto, Meterpreter e Havoc) a técnica apresentou 71,43% de efetividade no contorno dos monitoramentos e alertas, de forma que dos 42 itens testados, 5 tiveram algum tipo de falha na execução e/ou alerta após a execução e 7 apresentaram falha na execução com bloqueio total do processo.

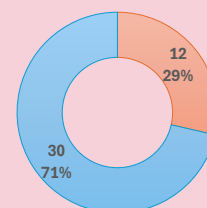
Isolando estes números por efetividade dos produtos, 57,14% dos produtos foram efetivos em identificar e/ou bloquear o ataque empreendido.

Durante os testes da execução do Metasploit Open Source [16] foi presenciado bloqueios e alertas na execução de alguns comandos após o estabelecimento da sessão do metasploit. Este comportamento de identificação e bloqueio é esperado, pois muitos destes comandos executam outros processos do windows, e como os novos processos (mesmo que filhos do processo do HookChain) não terão os implantes de bypass realizados pelo HookChain o EDR será capaz de monitorar estes comportamentos e realizar as devidas ações mitigatórias.

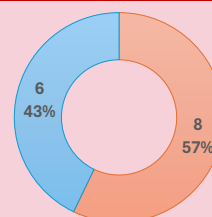
Porém com a utilização do Framework Havoc [17] um número menor de bloqueios foi presenciados, demonstrando, desta forma, que as identificações e possíveis bloqueios estão diretamente atrelados as ações realizadas bem como o Framework utilizado.

Possivelmente com a utilização de outros produtos com comportamento mais furtivo como o próprio Metasploit Pro, CobaltStrike entre outros, a maioria das ações realizadas passarão despercebidas.

Resultado 4: Considerando os pontos isolados de testes realizados com usuário de baixo privilégio, em 71,43% dos testes realizados (30 de 42) obtiveram sucesso sem nenhuma identificação ou bloqueio, em outras palavras, a execução do contorno da camada de segurança foi realizada com sucesso.



Resultado 5: 57,14% das soluções de EDR analisadas (8 de 14) foram capazes de identificar ou bloquear a ação realizada.



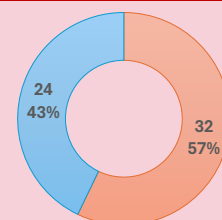
5.2.2. Acesso administrativo

Diferentemente como nos testes com usuário de baixo privilégio, nos testes com permissão administrativa, onde o objetivo foi obter as credenciais salvas em memória do processo LSASS, os produtos apresentaram uma melhor efetividade, de forma que conseguiram conter ou alertar em 57,14% dos cenários testados.

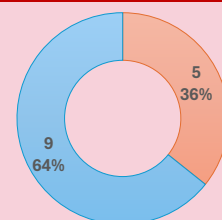
Este procedimento foi escolhido exatamente em virtude de ser um processo extremamente crítico dentro da arquitetura do sistema operacional, e suas técnicas de extração se baseiam em procedimentos extremamente conhecidos e utilizados por diversos malwares e artefatos utilizados em testes de invasão.

Percebeu-se também, durante este teste, que quando utilizada uma estratégia nova como foi o caso do teste onde utilizamos um LSASS dumper 100% escrito em Assembly por mim, a efetividade dos produtos na defesa e identificação caiu para 35,7%.

Resultado 6: Considerando os pontos isolados de testes realizados com usuário administrativo, em 57,14% dos testes realizados (32 de 56) foram mitigados, com ou sem alerta. Desta forma 24 dos 56 testes obtiveram sucesso sem nenhuma identificação ou bloqueio.



Resultado 7: Mesmo trabalhando com análise comportamental os produtos de EDR/XDR ainda não tiveram condições de identificar e mitigar ataques customizados e mais sofisticados, tendo desta forma a técnica de HookChain uma efetividade no contorno de 64,29% neste cenário.



6. CONCLUSÃO

O HookChain se mostrou uma técnica efetiva para a realização dos contornos das camadas de segurança aplicadas pelos produtos de EDR, tendo efetividade de até 71% de bypass nos testes realizados, chegando a contornar 100% das identificações de alguns produtos avaliados.

Como o tema “bypass de defesas” é um assunto extremamente vivo, e com cantantes mudanças tanto nas cadeias e técnicas de ataque, bem como nos processos de defesa, os resultados aqui apresentados representam uma foto do momento em que foram realizados, de forma que não há como garantir ou prever o comportamento das soluções após a aplicação das devidas correções por parte dos fabricantes bem como por melhorias e falhas nas configurações particulares de cada ambiente implementado.

Nota: Alguns fabricantes como a SentinelOne e Trend me procuraram após a finalização deste estudo solicitando apoio para o entendimento e subsequente melhoria nas identificações e telemetrias dos seus respectivos produtos.

7. AGRADECIMENTOS

Eu não poderia finalizar este trabalho sem agradecer primeiramente a Deus por sempre estar a frente de todas as minhas batalhas. Mas também a minha esposa, fonte de inspiração, mulher guerreira que sempre caminha ao meu lado me apoiando e incentivando a cada dia ser 1% melhor.

Aos meus amados filhos também o muito obrigado, pois a cada abraço, a cada carinho me sinto o homem mais importante do mundo, e sei o quanto as horas dedicadas a este trabalho fazem falta para cada um de nós.

Não poderia deixar de citar os meus pais que sempre se desdobraram para me dar o melhor estudo, carinho e suporte alçar voos galgando os meus sonhos, mesmo que isso significasse ir para longe.

E para finalizar a toda a comunidade de Cyber Segurança do Brasil e em especial a todos aqueles que confiaram os seus ambientes, licenças e máquinas virtuais para que os testes pudessem ser conduzidos, sem vocês certamente este trabalho estaria incompleto.

De modo muito especial gostaria de citar aqui alguns nomes que fizeram e fazem a diferença em toda a minha vida com sua amizade, puxões de orelha, apoio quando fraquejo e o mais importante, com as suas orações. Muito obrigado, Eder Luis, Marcus Prestes, Paulo Trindade, Aroldo Chociai e Rafael Salema, vocês fazem parte dessa história.

8. BIBLIOGRAPHY

- [1] M. Hand, *Evading EDR: The Definitive Guide to Defeating Endpoint Detection Systems*, San Francisco: No Starch Press, Inc, 2024.
- [2] M. Russinovich, D. A. Solomon e A. Lonescu, *Windows Internals*, Sixth Edition, Part 1, Redmond, Washington: Microsoft Press, 2012.
- [3] P. Yosifovich, A. Lonescu, E. M. Russinovich e A. D. Solomon, *Windows Internals Seventh Edition - Part 1*, Redmond: Microsoft Press, 2017.
- [4] Microsoft, "Microsoft Learn," [Online]. Available: <https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170>. [Acesso em 21 03 2024].
- [5] Microsoft, "PE Format," [Online]. Available: <https://docs.microsoft.com/windows/win32/debug/pe-format>. [Acesso em 21 03 2024].
- [6] NtCore, "Explorer Suite," [Online]. Available: https://ntcore.com/?page_id=388. [Acesso em 21 03 2024].
- [7] R. Batra, "API Monitor," [Online]. Available: <http://www.rohitab.com/apimonitor>. [Acesso em 21 03 2024].
- [8] M. Pietrek, *Windows 95 System Programming Secrets*, Foster City: IDG Books Worldwide, Inc, 1995.
- [9] @modexpblog, "Bypassing User-Mode Hooks and Direct Invocation of System Calls for Red Teams," [Online]. Available: <https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/>. [Acesso em 22 03 2024].
- [10] am0nsec; smelly__vx;, [Online]. Available: <https://vxug.fakedoma.in/papers/VXUG/Exclusive/HellsGate.pdf>. [Acesso em 22 03 2024].
- [11] Reenz0h from Sektor7, 23 04 2021. [Online]. Available: <https://blog.sektor7.net/#!/res/2021/halosgate.md>. [Acesso em 22 03 2024].
- [12] C. Joca, "NtGate, An implementation of Halo's Gate and indirect syscalls," [Online]. Available: <https://github.com/hiatus/NtGate>. [Acesso em 22 03 2024].
- [13] H. C. J. M4v3r1ck, "HookChain Research - Step 1," [Online]. Available: <https://github.com/helviojunior/hookchain/tree/0b4a953c10a18f53aa68f7588db9818730dd7a52>. [Acesso em 22 03 2024].
- [14] S. Fewer, "ReflectiveDLLInjection," [Online]. Available: <https://github.com/stephenfewer/ReflectiveDLLInjection/>. [Acesso em 22 03 2024].
- [15] Gartner, "Magic Quadrant for Endpoint Protection Platforms," 31 12 2023. [Online]. Available: <https://www.gartner.com/doc/reprints?id=1-2G7RNK65&ct=240112&st=sb>.
- [16] Rapid7, [Online]. Available: <https://www.metasploit.com/>. [Acesso em 03 04 2024].
- [17] C5pider, [Online]. Available: <https://havocframework.com/>. [Acesso em 03 04 2024].