|  |  |
| --- | --- |
| CURSO DE CIÊNCIA DA COMPUTAÇÃO – TCC | |
| (   ) PRÉ-PROJETO     ( X ) PROJETO | ANO/SEMESTRE: 2024/1 |

**HyperMon – Ferramenta para Monitoramento de Chamadas de Sistema com uso de Driver e Hypervisor**

João Vitor de Oliveira

Prof. Danton Cavalcanti Franco Junior - Orientador

# Introdução

Segundo CISO ADVISOR (2023), o número de ataques cibernéticos registrados no ano de 2022 globalmente foi 38% maior que o volume total observado no ano de 2021. Devido a este fato, se enfrenta um desafio cada vez maior relacionado à segurança da informação. Ameaçando a integridade, a confidencialidade e a disponibilidade dos sistemas de informação em todo o mundo. Nesse contexto de acordo com a Kaspersky (2013) um dos principais vetores de ataque que tem preocupado a comunidade de segurança cibernética é o uso de *rootkits*, que muitas vezes se infiltram no núcleo do sistema operacional, explorando técnicas avançadas de *hooking* para comprometer a segurança e a privacidade dos sistemas.

Ainda segundo a Kaspersky (2013), historicamente os *rootkits* eram frequentemente desenvolvidos com o intuito de se esconderem no sistema, tornando-se virtualmente invisíveis para as soluções de segurança convencionais. Uma das técnicas mais utilizadas para esse fim era o *hooking* da System Service Dispatch Table *(*SSDT), permitindo que os *rootkits* interceptassem chamadas de sistema (*syscall*) e, assim, obtivessem controle sobre o sistema operacional. O desenvolvimento dessa técnica representou um sério desafio para a segurança cibernética, pois tornava extremamente difícil a detecção e a mitigação de ameaças avançadas (Salinas, 2023).

De acordo com Reginato (2019), à medida que a ameaça dos *rootkits* cresceu, também evoluiu a resposta da indústria de segurança cibernética. Um dos marcos nesse avanço foi a introdução do mecanismo de defesa conhecido como PatchGuard (PG) ou Kernel Patch Protection (KPP), desenvolvido pela Microsoft. O PG tornou-se uma barreira eficaz contra os *rootkits* que exploravam o *hooking* da SSDT, protegendo o núcleo do sistema operacional e garantindo a integridade das chamadas de sistema.

Diante de todas as proteções oferecidas por ferramentas de antivirus e *Endpoint* *Detection* *and* *Response* (EDR), os *malwares* têm buscado novas técnicas para dificultar o processo de defesa. Uma das técnicas utilizadas é a de obfuscação e virtualização combinadas com *direct syscall* que torna ainda mais dificil o processo de rastreio da origem da chamada (Sikander, 2022).

Tendo em vista os fatos citados, o corrente trabalho propõe uma ferramenta que comunicará com um *driver* em *kernelmode* possibilitando o monitoramento ativo de chamadas de sistema no escopo do sistema (não limitado somente ao escopo da aplicação) utilizando de técnicas de *hooking*.

## OBJETIVOS

Este trabalho tem como objetivo utilizar de técnicas de *hooking* e virtualização para a interceptação de chamadas do sistema para facilitar o processo de engenharia reversa, principalmente em aplicações obfuscadas ou virtualizadas.

Os objetivos específicos são:

1. criar uma ferramenta eficiente e confiável que seja capaz de monitorar chamadas de sistema em tempo real em aplicações obfuscadas ou virtualizadas em ambientes Windows;
2. catalogar e identificar chamadas de sistema críticas que são comumente usadas em aplicações obfuscadas ou virtualizadas, fornecendo um foco específico para o monitoramento;
3. implementar funcionalidades para analisar o comportamento das aplicações em execução, visando detectar padrões anômalos ou suspeitos relacionados ao uso de chamadas de sistema;
4. desenvolver uma interface de usuário que permita a fácil configuração e visualização dos dados coletados, bem como a geração de relatórios detalhados das atividades monitoradas;
5. realizar testes extensivos para validar a eficácia e confiabilidade da ferramenta em diferentes cenários e configurações de sistema operacional Windows.

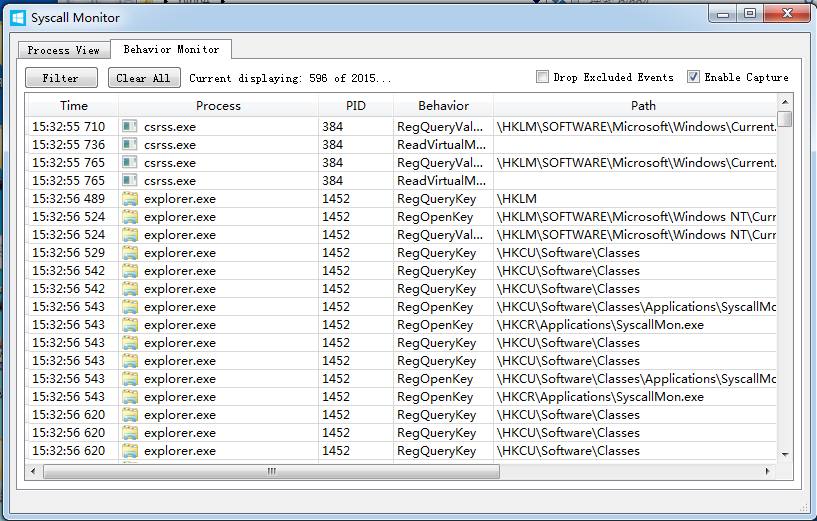
# trabalhos correlatos

Nesta seção são abordados trabalhos com características similares aos principais objetivos do presente estudo. A subseção 2.1 apresenta uma ferramenta para análise de execução de chamadas de sistema com uma interface gráfica (Wang, 2017), a subseção 2.2 introduz a utilização de *process instrumentation callback* para obter notificações de chamadas executadas na camada de usuário (Deputation, 2021). Por fim, a subseção 2.3 refere-se a uma ferramenta desenvolvida pela Microsoft frequentemente utilizada pela área de DevOPS (Microsoft, 2021).

## SYSCALL-MONITOR

Desenvolvido por Wang (2017), é uma contribuição significativa para a área de segurança da informação e monitoramento de chamadas de sistema em sistemas Windows. Este projeto utiliza as tecnologias de virtualização da Intel, Virtualization Technology (VT-X) e Extended Page Tables (EPT). Isso permite a criação de *stealth hook*, que se refere a uma técnica especializada e avançada de interceptação e monitoramento de chamadas de sistema em um sistema operacional, como o Windows. Essencialmente, essas tecnologias permitem que o Syscall-Monitor observe e registre as atividades do sistema de forma discreta, sem chamar a atenção das medidas de segurança convencionais (Wang, 2017). Em outras palavras, os *stealth hooks* agem como ganchos ou pontos de observação que são inseridos no sistema de forma oculta, permitindo que o Syscall-Monitor rastreie e analise as interações entre aplicativos e o kernel do sistema operacional sem ser detectado, proporcionando assim uma visão profunda do comportamento do sistema para fins de segurança e monitoramento avançados. A Figura 1 abaixo apresenta a interface gráfica da aplicação.

Figura 1 – Syscall Monitor



Fonte: Wang (2017).

Wang (2017) utiliza a tecnologia de virtualização da Intel que permite a criação de máquinas virtuais e oferece suporte para recursos avançados, como a tradução de endereços de segundo nível, que é essencial para o funcionamento do projeto. A principal funcionalidade do Syscall-Monitor é adicionar uma camada adicional de tradução entre os endereços virtuais e os endereços reais usados pelo sistema operacional Windows. Isso possibilita a modificação de rotinas nativas do *kernel* do Windows sem ser detectado pelo PG.

De acordo com Wang (2017), são utilizados *stealth hooks* que permitem que o Syscall-Monitor consiga monitorar e interceptar chamadas de sistema de forma discreta, o que é essencial para aplicações de segurança e monitoramento avançadas. Isso torna possível a análise do comportamento do sistema, identificação de atividades suspeitas e até mesmo possibilita obter os parâmetros da rotina, visto que a *stack* ainda não foi limpa, sem acionar alarmes de segurança.

Em resumo, o projeto Syscall-Monitor de Wang (2017) se mostra eficaz na detecção de chamadas de sistema em todo o sistema (*system-wide)*, utilizando de tecnologias de virtualização VT-X e EPT da Intel. Contudo, o projeto apresenta a limitação de não ser compatível com processadores da fabricante AMD. Essa limitação se dá ao fato das diferenças fundamentais na implementação das tecnologias de virtualização entre ambas as fabricantes, uma vez que cada uma possui funcionalidades específicas e devem ser implementadas conforme o manual (Wang, 2017).

## INSTRUMENTATION\_CALLBACKS

Segundo Deputation (2021), o método empregado no projeto Instrumentation\_callbacks é baseado na utilização de *callbacks* de instrumentação de processo. Essas *callbacks* permitem o monitoramento de eventos específicos, como a execução de *syscalls* dentro do espaço de memória de um processo. Isso é realizado através da injeção de uma Dynamic Link Library (DLL) no espaço de memória do processo em questão, permitindo que a DLL registre e responda a eventos específicos.

Ainda de acordo com Deputation (2021), as *callbacks* de instrumentação de processo oferecem a capacidade de interceptar e registrar a execução de *syscalls*, o que pode ser extremamente valioso na detecção de atividades maliciosas ou comportamentos anômalos em processos em execução. Essa abordagem é particularmente útil em cenários de segurança cibernética, onde a identificação de *syscalls* suspeitas pode ser um indicativo importante de atividades maliciosas.

No entanto, é importante destacar algumas limitações inerentes a esse método de detecção. A principal limitação é que ele exige a injeção de uma DLL no espaço de memória do processo que está sendo monitorado. Isso pode ser problemático em ambientes onde a injeção de DLLs é considerada uma atividade suspeita ou maliciosa. Além disso, a injeção de DLLs pode ser bloqueada ou detectada por medidas de segurança avançadas. Também é importante notar que a *stack* é limpa antes de executar a *callback*, ou seja, os paramêtros que foram informados para a execução da rotina não estarão mais disponíveis (Deputation, 2021).

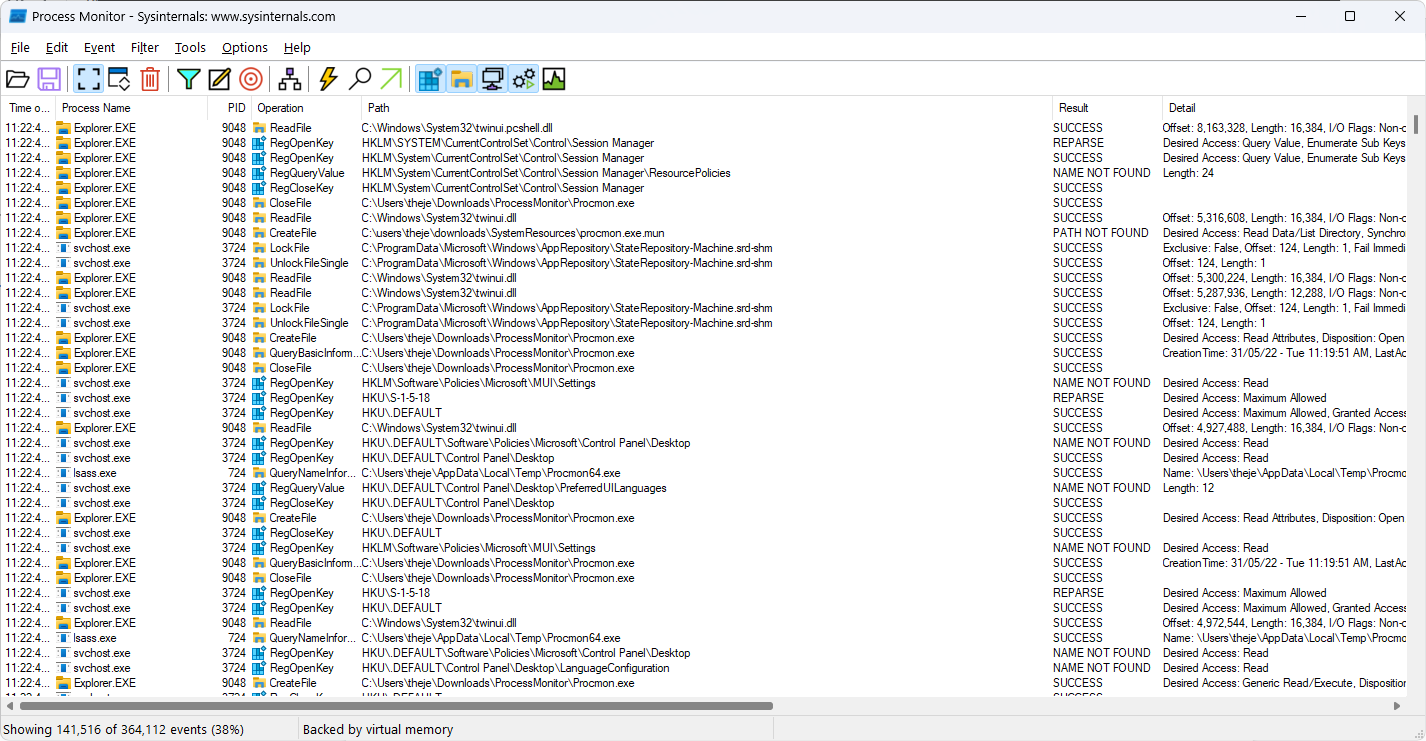
Outra consideração importante é que, como qualquer método de detecção, ele não é imune à possibilidade de ser burlado. Um atacante que tenha conhecimento sobre a presença da DLL de instrumentação no processo monitorado pode tentar desativar ou burlar as *callbacks* de instrumentação, comprometendo assim a eficácia da detecção de *syscalls* (Deputation, 2021).

Em resumo, o projeto Instrumentation\_callbacks de Deputation (2021) apresenta uma forma de detectar execução de *syscalls* por meio de *callbacks* de instrumentação de processo, sem a necessidade da instalação de um *driver* para efetuar a monitoração, entretanto, é importante destacar as limitações inerentes deste método, visto que é possível desabilitar as *callbacks* temporariamente e passar despercebido e também na execução da *callback* a *stack* já foi limpa, não sendo possível obter os parâmetros da rotina.

## PROCESS MONITOR

Segundo Marin (2022), o Process Monitor é uma ferramenta desenvolvida pela Microsoft, utilizada para monitoramento e diagnóstico de processos amplamente reconhecida no ecossistema Windows. É possível visualizar sua interface na Figura 2.

Figura 2 – Getting started with Procmon: The Beginner’s Guide to Monitoring Windows Systems



Fonte: Marin (2022).

Ainda segundo Marin (2022), o método empregado pelo Process Monitor para monitorar processos envolve a operação em modo *kernel*, onde um *driver* é carregado no sistema operacional. Esse *driver* atua como uma camada intermediária que permite ao Process Monitor registrar todas as atividades relevantes. Uma das funcionalidades do Process Monitor é a capacidade de realizar o *dumping* (registro) de *callstacks*, permitindo que os usuários localizem com precisão o ponto de origem de uma chamada de sistema (*syscall*) dentro de um aplicativo em execução (Marin, 2022).

Em resumo, é importante destacar que, apesar de sua eficácia e compatibilidade com ambas as fabricantes de processadores, o Process Monitor não é capaz de capturar todas as *syscalls* por meio de *callbacks* registradas no sistema operacional. Algumas chamadas de sistema podem não ser visíveis por meio deste método, o que limita parcialmente sua capacidade de monitorização. Essa limitação muitas vezes é necessária para estar em conformidade com o PG, também é importante citar que o Process Monitor não é um projeto *open-source* desta forma, não tem flexibilidade para modificação (Marin, 2022).

# proposta

A seguir é apresentada a justificativa para o desenvolvimento da ferramenta, juntamente com os principais requisitos e a metodologia de desenvolvimento a ser utilizada. Também são relacionados os assuntos e as fontes bibliográficas que irão fundamentar o estudo presente.

## JUSTIFICATIVA

Segundo Sikander (2022) o processo de detecção de chamadas de sistemas realizadas de forma manual em aplicações obfuscadas ou virtualizadas é dificultado, pois há a construção de um *bytecode* customizado, fazendo com que este seja somente interpretado pela Virtual Machine (VM). Então, neste contexto, é proposto um projeto para desenvolver uma ferramenta avançada de monitoramento de chamadas de sistema com suporte às tecnologias de virtualização VT-X da Intel e AMD-V da AMD para estar dentro das conformidades do mecanismo de defesa PG, contribuindo ativamente no processo de rastreabilidade de chamadas de sistema facilitando o processo de análise dinâmica comumente destinado aos analistas de malware.

Conforme constatado no Quadro 1, a aplicação Syscall-Monitor por Wang (2017) apenas utiliza da tecnologia de virtualização VT-X. O projeto Instrumentation\_callbacks por Deputation (2021) utiliza de *callback* de instrumentação de processo, sendo assim, apenas reside na camada de usuário (usermode) do sistema operacional, tornando mais passível de desativação por alguma outra aplicação comum, visto que ambas estarão com os mesmos privilégios. Já o Process Monitor está presente na camada do *kernel* (núcleo) do sistema operacional, havendo a limitação onde nem todas as rotinas que estão presentes na SSDT constam para o registro de uma *callback*.

Quadro 1 - Comparativo dos trabalhos correlatos

|  |  |  |  |
| --- | --- | --- | --- |
| Trabalhos Correlatos  Características | Wang (2017) | Deputation (2021) | Marin (2022) |
| Técnica utilizada | *Hooking* com virtualização *VT-X / EPT* | *Callback* de instrumentação de processo | *Callbacks* registradas no *kernel* do sistema operacional |
| É possível obter os parâmetros da rotina | Sim | Não | Sim |
| Executado na camada | *Kernel* | Usuário | *Kernel* |
| É possível burlar através da camada de usuário | Não | Sim | Não |
| Compatível entre os fabricantes | Intel | Intel e AMD | Intel e AMD |
| Possui um driver centralizado de fácil manutenção | Não | Não | Não |

Fonte: elaborado pelo autor.

Como a ferramenta de Wang (2017) utiliza diretamente de *hooking* para a interceptação das chamadas de sistema, neste momento ainda não houve a limpeza da memoria *stack*, sendo assim, ainda é possível obter os parâmetros que foram informadas para a rotina. Já o projeto Instrumentation\_callbacks como previamente detalhado por Deputation (2021) no momento que haver a chamada para essa *callback* a *stack* já foi limpa, impossibilitando obter os parâmetros de execução da função em questão. Por fim, a ferramenta Process Monitor é capaz de obter os parâmetros da rotina devido a sua ordem de chamada ser antes da execução em si da rotina com a única limitação de um escopo reduzido.

Previamente dito, a ferramenta de Deputation (2021) Instrumentation\_callback reside na camada de usuário, sendo assim, ela é a mais comum, é a camada onde as aplicações tradicionais são alocadas facilitando o atacante burlar por um executável ou DLL. Segundo Wang (2017) como hypervisors dependem da implementação de cada CPU, o Syscall-Monitor não foi implementado para a tecnologia AMD-V impossibilitando o uso por parte dos usuários de processadores da fabricante AMD.Além disso, o Syscall-Monitor utiliza o framework de virtualização ddimon por Tanda (2016), no qual Tanda (2016) utiliza de mais alguns outros frameworks para a construção do seu projeto, fazendo com que o processo de manutenção do Syscall-Monitor seja mais complicado. Já o Process Monitor não é um projeto de código aberto impossibilitando a criação de novas funcionalidades (Marin, 2022).

Wang (2017) também relata uma lentidão causada proveniente da má otimização no uso de memória, sendo assim, é necessário reiniciar a máquina depois de certo tempo de uso. Já o projeto Instrumentation\_callbacks por Deputation (2021) como utiliza da camada de usuário o máximo que poderá acontecer é houver um *crash* na aplicação. E por fim, o Process Monitor por ser um projeto maduro não apresenta o problema citado.

Com isso, espera-se que a aplicação desenvolvida seja capaz de monitorar as chamadas de sistema (*syscall*) a nível de sistema (*system-wide*) compatível com tecnologia de virtualização AMD, que possibilite que o usuário obtenha os parâmetros de execução, tempo de execução e também possua uma interface gráfica para fácil visualização dos dados.

## REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A ferramenta a ser desenvolvida deverá:

1. identificar as chamadas de sistema (*syscall*) executadas de forma manual (fora da ntdll) (Requisito Funcional – RF);
2. apresentar o *dumping* de *callstack* com os nomes de módulos presentes no processo e calcular o Relative Virtual Address (RVA) (RF);
3. apresentar o tempo de duração da execução da chamada de sistema (*syscall*) (RF);
4. ser compatível entre os principais fabricantes de processadores do mercado (Intel e AMD) (Requisito Não Funcional - RNF);
5. ser compatível com as *builds* recentes do Windows 10 e Windows 11 (RNF);
6. ser performático de modo que o usuário não sofra com problemas de desempenho impossibilitando o uso da máquina (RNF);
7. construir um *driver* centralizado que facilite a manutenção das funcionalidades de *hooking* (RNF);
8. ser implementado utilizando a linguagem de programação C++ (RNF);
9. desenvolver utilizando a IDE Visual Studio Community (RNF).

## METODOLOGIA

O trabalho será desenvolvido observando as seguintes etapas:

1. levantamento bibliográfico: pesquisar e estudar sobre desenvolvimento *kernel mode,* técnicas de *hooking*, virtualização e trabalhos correlatos;
2. levantamento dos requisitos: levando em consideração as informações da etapa anterior, validar os requisitos propostos para a aplicação;
3. desenvolvimento de driver centralizado: implementar uma estrutura de centralização para facilitar o desenvolvimento da ferramenta e de sua manutenção;
4. modelagem do driver de *hooking:* após o desenvolvimento do item (c), desenvolver o *driver* que conterá as rotinas nativas do *kernel* do sistema operacional e comunicará a aplicação sobre a execução de uma nova chamada de sistema (*syscall*);
5. implementação de interface gráfica: verificar se a linguagem de programação C++ atenderá os principais requisitos para a implementação da interface gráfica e desenvolver;
6. testes: avaliar a performance, assim como verificar se está sendo realizado todos os *hookings* propostos para a ferramenta e verificar sobre as compatibilidades entre os sistemas operacionais e processadores, por fim checar a segurança do uso da aplicação.

As etapas serão realizadas nos períodos apresentados no Quadro 2.

Quadro 2 - Cronograma de atividades a serem desensolvidas

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2024 | | | | | | | | | |
|  | jul. | | ago. | | set. | | out. | | nov. | |
| etapas / quinzenas | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 |
| levantamento bibliográfico |  | X | X |  |  |  |  |  |  |  |
| levantamento dos requisitos |  |  | X |  |  |  |  |  |  |  |
| desenvolvimento do *driver* centralizado |  |  |  | X |  |  |  |  |  |  |
| modelagem do *driver* de *hooking* |  |  |  |  | X | X | X |  |  |  |
| implementação da interface gráfica |  |  |  |  | X | X | X | X |  |  |
| testes |  |  |  |  |  |  |  |  | X | X |

Fonte: elaborado pelo autor.

# REVISÃO BIBLIOGRÁFICA

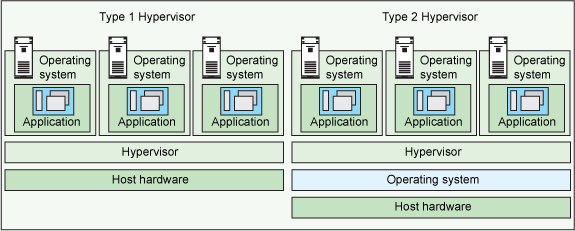
Esta seção descreve brevemente as tecnologias relevantes para o presente estudo que são Hypervisors, Chamadas de Sistema (*syscall*) e técnicas de *hooking.*

## Hypervisors

Segundo Martin (2017), Hypervisors fazem parte da ampla área da computação conhecida como virtualização, um conceito que existe há certo tempo. No meio da década de 1960, os *mainframes* dominavam o mundo da computação corporativa. No entanto, segundo Martin (2017), esses *mainframes* eram relativamente raros e extremamente caros. Muitas das vezes serviam multiplos usuários, que compartilhavam uma fatia do tempo para executar alguns trabalhos. Isso significa que, do ponto de vista de cada usuário, parecia que eles tinham um *mainframe* exclusivamente destinado a eles. Neste contexto, um Hypervisor é um *software* que gerencia o *hardware* disponível no computador em que está sendo executado para os demais sistemas operacionais que está sendo controlado (*guests*).

De acordo com Martin (2017), existem dois tipos de virtualização *baremetal* (type-1) e *guest* (type-2). A virtualização *baremetal*, é uma técnica na qual um Hypervisor é instalado diretamente no *hardware* físico de um servidor, sem a necessidade de um sistema operacional hospedeiro (*host*). Isso significa que o Hypervisor funciona como sistema operacional primário, controlando o acesso direto aos recursos físicos da máquina e permitindo a execução de múltiplas Virtual Machines (VMs) independentes. Já de acordo com Martin (2017), um *Hypervisor* do tipo *guest* é um *software* de virtualização que é instalado e executado sobre um sistema operacional *host* existente. Através da Figura 3 é possível observar os tipos de virtualização disponíveis e seu funcionamento.

Figura 3 – Learn about hypervisors, system virtualization, and how it works in a cloud environment



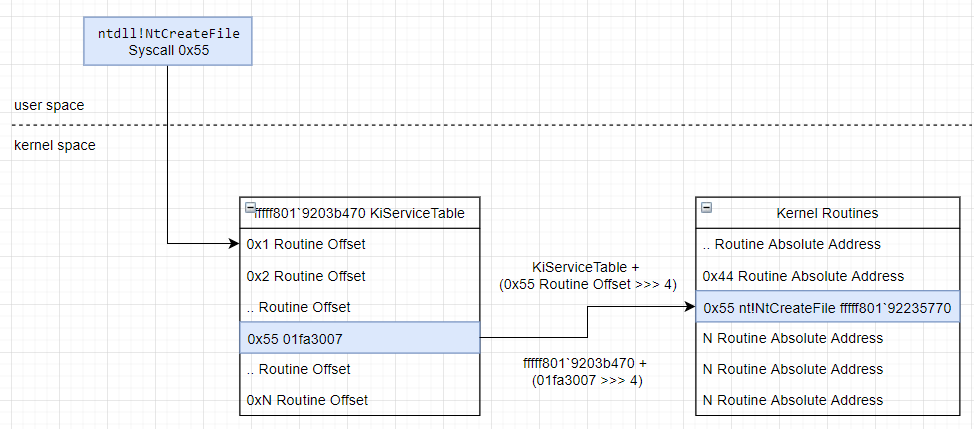
Fonte: Tolheti (2011).

Segundo a TechTarget (2014), Hypervisors permitem uma tecnologia chamada de Second Level Address Translation(SLAT) que reduz o *overhead* causado por conta de VM-Exits. Isto é possível pois é feita uma nova camada de tradução de Guest Physical Address(GPA) em Host Physical Address(HPA). Com isto o Core Processor Unit (CPU)possui referência para uma tabela de páginas chamada de Page Tables, que fará a tradução a nível de *hardware*, ou seja, oHypervisor não é chamado para realizar estas traduções. Na Intel, a tecnologia SLAT é conhecida como EPT e na AMD é comumente encontrada como Rapid Virtualization Indexing *(*RVI) ou Nested Page Tables (NPT).

## Chamada de sistema (*syscall*)

De acordo com Brizendine (2023), uma *syscall* é feita de algumas funções na biblioteca da NTDLL como uma maneira de requisitar um serviço do *kernel* do sistema operacional. Uma *syscall* é o último passo entre a camada de usuário e a camada de *kernel* e uma *syscall* não é feita para ser usada por programadores, sendo assim, programadores irão chamar outras funções, geralmente provindas da biblioteca KERNEL32. Uma *syscall* também utiliza um número especial de serviço, nomeado de System Service Number (SSN), que, quando é executada, tem o seu número atribuído ao registrador EAX. No momento da execução, a rotina presente no *kernel* é obtida através da resolução da SSN utilizando a SSDT, que é a tabela responsável por conter todas as rotinas que poderão ser requisitadas através da instrução *SYSCALL* (Ired Team, 2019). Sendo assim, através da Figura 4 abaixo é possível observar o fluxo de resolução do endereço nativo da rotina através da *SSDT*.

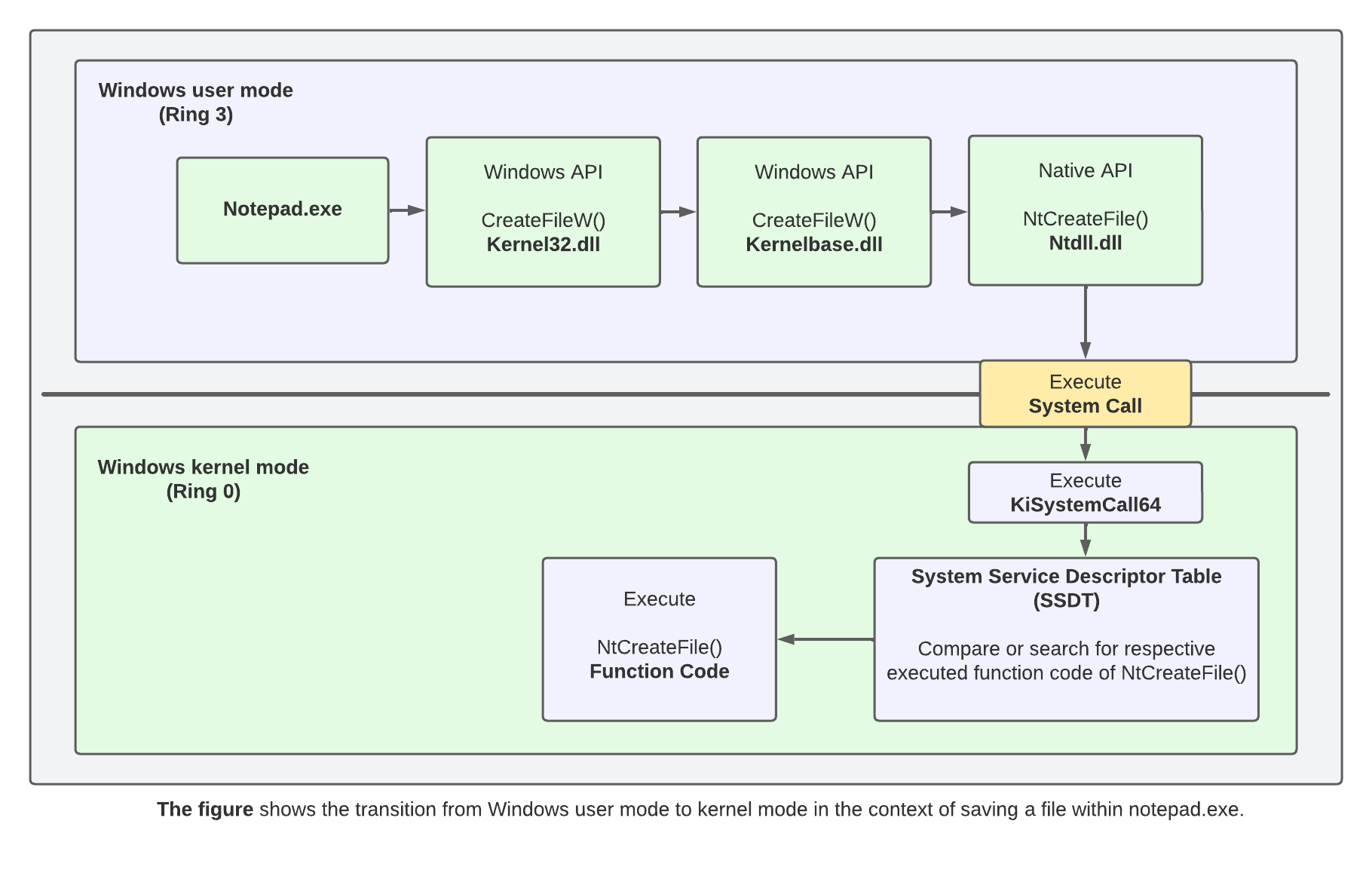
Figura 4 – System Service Descriptor Table



Fonte: Ired.Team (2020).

De acordo com a RedOps (2024) existem duas principais formas de execução de uma *syscall,* indireta e direta. Quando uma chamada é executada de forma indireta, a mesma utiliza de intermediários ao invés de chamar a função diretamente. Como as *syscalls* são interfaces de baixo nível que os programas utilizam para solicitar serviços (operações de arquivo) para o núcleo do sistema, como alocações de memória e operações de rede. Quando é executada indiretamente, a *syscall* é chamada através de outras rotinas, que por sua vez fazem a abstração do processo. Por exemplo, na criação de um arquivo é chamada a função CreateFile que está localizada na biblioteca KERNEL32.DLL, que por sua vez fará o desvio para a rotina NtCreateFile localizada na NTDLL.DLL. Por fim a NTDLL executa a instrução *syscall* que realiza a transição entre *usermode* para *kernelmode*, sendo possível observar na Figura 5 abaixo:

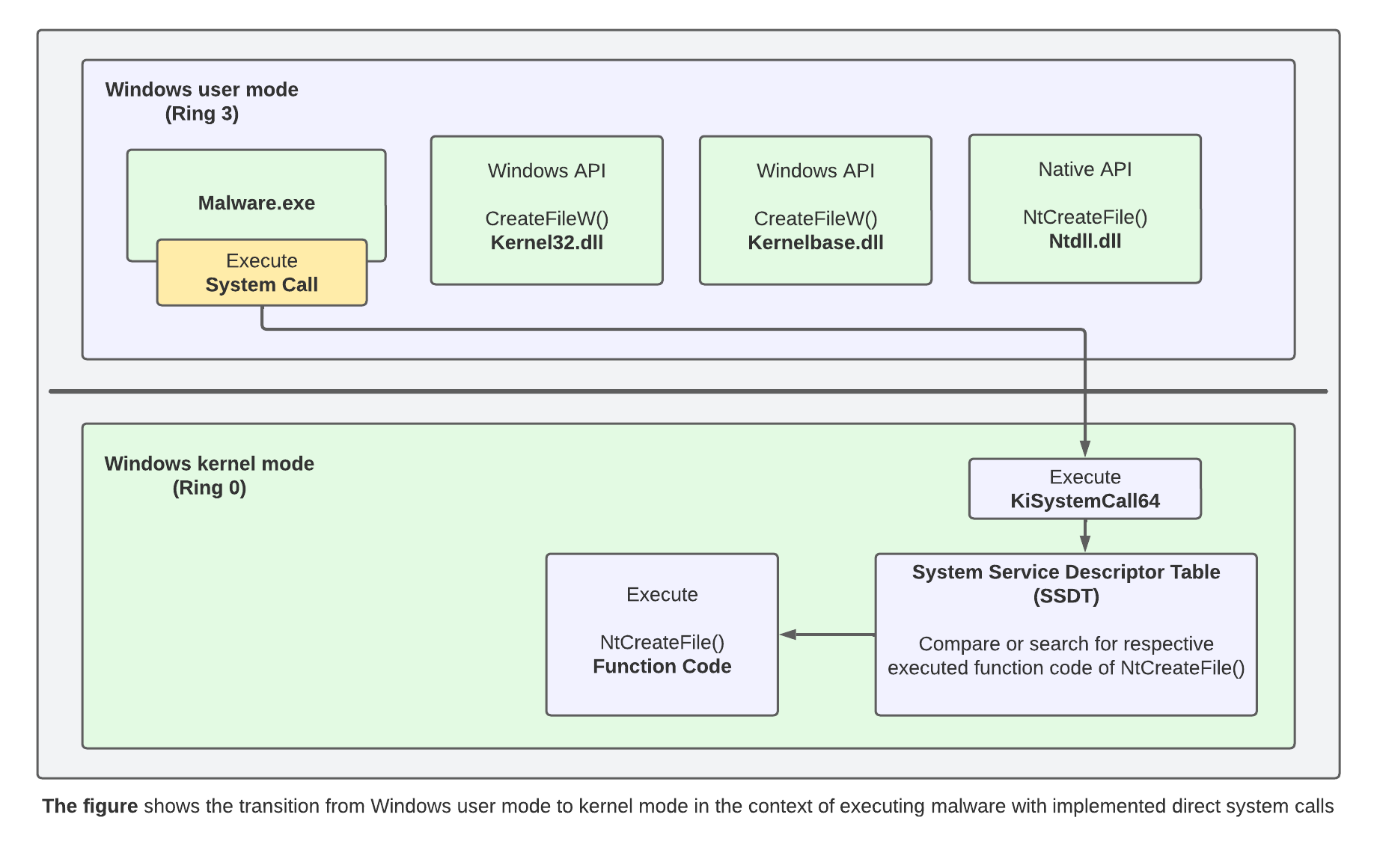
Figura 5 – Indirect Syscall



Fonte: RedOps (2024).

Ainda segundo a RedOps (2024), a técnica de *syscall* direta consiste na chamada da instrução *syscall* a partir de uma *stub* alocada no processo, neste caso ela não é chamada a partir da NTDLL.DLL, são usadas para contornar camadas intermediárias, pois ao invés de chamar uma função nativa que pode ser monitorada ou alterada por uma solução de antivírus ou EDR, o código malicioso constrói manualmente a estrutura necessária para a invocação da instrução e a invoca diretamente, sendo assim, elas não são detectavéis através de *hooking* na camada de usuário (*usermode)*. Seu fluxo pode ser visto na Figura 6 abaixo.

Figura 6 – *Direct Syscall*

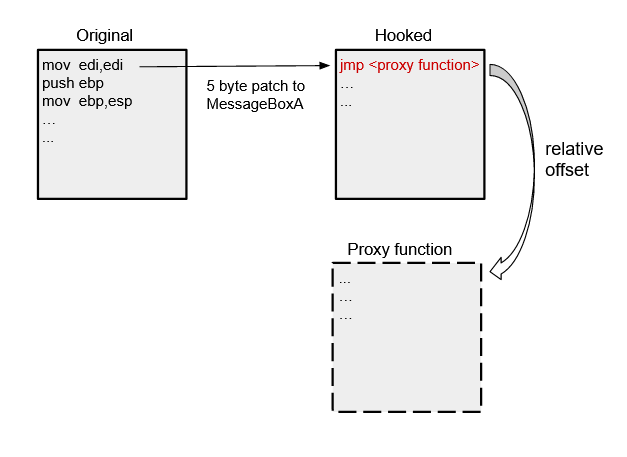


Fonte: RedOps (2024).

## Técnica de *hooking*

Segundo Shaid e Maarof (2015), a técnica de *hooking* é uma técnica que permite monitorar, modificar ou estender o comportamento de funções ou rotinas de uma Application Programming Interface (API) em um *software* ou sistema operacional. Essa técnica é frequentemente utilizada para aprimorar a funcionalidade de programas, adicionar recursos de segurança, rastrear a atividade do sistema ou até mesmo depurar o *software* em questão. O uso de *hooking* é poderoso, mas também apresenta desafios, já que pode ser abusado para atividades maliciosas, como *spyware* ou *malware*. Portanto, é importante que o uso dessa técnica seja ético e legal. Muitos sistemas operacionais e aplicativos de segurança possuem medidas para detectar atividades suspeitas sobre o uso desta técnica. Abaixo na Figura 7 é apresentado uma breve ilustração de como o desvio de fluxo via *hooking* funciona.

Figura 7 – Basic Windows API Hooking



Fonte: Hurst (2021).

Ainda segundo Shaid e Maarof (2015), existem diferentes tipos de hooking como o método de sobrescrever a Import Address Table (IAT), que é uma tabela contida em um executável do formato Portable Executable (PE), responsável por conter endereços de funções importadas de outras bibliotecas, como DLLs no sistema Windows. Quando um programa necessita chamar uma função externa, ele consulta a IAT para encontrar o endereço da função. Ao modificar os endereços desta tabela, é possível redirecionar as chamadas dessas funções para outros endereços, permitindo a interceptação e mudança do comportamento do programa.

Também de acordo com Shaaid e Maarof (2015), são utilizados *debug hooks* para a interceptação de um programa, permitindo que um depurador ou outro *software* de monitoramento interfira em seu funcionamento. Para a implementação dos mesmos, um *event handler* é registrado no processo alvo, que é responsável por capturar e processar as exceções geradas pela aplicação. Através de rotinas nativas do sistema operacional é possível forçar uma exceção de forma em que na hora de tratar seja sobrescrito o Extended Instruction Pointer (EIP) em aplicações de arquitetura x86 ou Register Instruction Pointer (RIP) em arquiteturas x64.

Ainda de acordo com Shaaid e Maarof (2015), existe a técnica de *inline hooking* que, diferente das outras técnicas previamente mencionadas, envolve a modificação direta do código de uma função alvo, sendo amplamente utilizada por projetos de monitoramento. Os primeiros passos são, identificar a função específica em que se deseja interceptar, salvar uma cópia dos *bytes* que correspondem a uma instrução da rotina pretendida, calcular o RVA em relação a função de interceptação e a função original para que posteriormente seja inserido uma instrução JMP que fará com que o fluxo de execução desvie para a rotina de interceptação. Por fim para concluir é feito novamente o desvio para um *backup* que contenha a instrução que foi sobrescrita e uma instrução JMP que irá restaurar o comportamento original.

Referências

BRIZENDINE, Bramwell **Windows Syscalls in Shellcode: Advanced Techniques for Malicious Functionality**. Amsterdam, [2023]. Disponível em: https://conference.hitb.org/hitbsecconf2023ams/materials/D1T2%20-%20Windows%20Syscalls%20in%20Shellcode%20-%20Advanced%20Techniques%20for%20Malicious%20Functionality%20-%20Bramwell%20Brizendine.pdf. Acesso em: 14 abr. 2024.

CISO ADVISOR. **Volume global de ciberataques teve alta de 38% em 2022**, [2023]. Disponível em: https://www.cisoadvisor.com.br/volume-global-de-ataques-ciberneticos-aumenta-38-em-2022. Acesso em: 14 abr. 2024.

DEPUTATION. **Instrumentation\_callbacks**, [2021]. Disponível em: https://github.com/Deputation/instrumentation\_callbacks. Acesso em: 14 abr. 2024.

HURST. **Basic Windows API Hooking**, [2021]. Disponível em: https://medium.com/geekculture/basic-windows-api-hooking-acb8d275e9b8. Acesso em: 14 abr. 2024.

IRED TEAM. **System Service Descriptor Table – SSDT**, [2019]. Disponível em: https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/glimpse-into-ssdt-in-windows-x64-kernel. Acesso em: 14 abr. 2024.

KASPERSKY. **O que é o Rootkit – Definição e Explicação**, [2013]. Disponível em: https://www.kaspersky.com.br/resource-center/definitions/what-is-rootkit. Acesso em: 14 abr. 2024.

MARIN. **Getting started with Procmon: The Beginner’s Guide to Monitoring Windows Systems**, [2022]. Disponível em: https://www.advancedinstaller.com/process-monitor-beginner-guide.html. Acesso em: 14 abr. 2024.

MARTIN. **Hypervisor Part 1 – What is a Hypervisor and How Does it Work?**, [2017]. Disponível em: https://blackberry.qnx.com/content/dam/qnx/whitepapers/2017/what-is-a-hypervisor-and-how-does-it-work-pt1.pdf. Acesso em: 14 abr. 2024.

OREANS. **Total Obfuscation against Reverse Engineering** [2024]. Disponível em: https://www.oreans.com/CodeVirtualizer.php. Acesso em: 14 abr. 2024.

REDOPS. **Direct Syscalls: A journey from high to low** [2024]. Disponível em: https://redops.at/en/blog/direct-syscalls-a-journey-from-high-to-low. Acesso em: 13 jun. 2024.

REGINATO. **Updated Analysis of PatchGuard on Microsoft Windows 10 RS4** [2019]. Disponível em: https://blog.tetrane.com/downloads/Tetrane\_PatchGuard\_Analysis\_RS4\_v1.01.pdf. Acesso em: 13 jun. 2024.

SALINAS. **Fantastic Rootkits: And Where To Find Them (Part 3) – ARM Edition** [2023]. Disponível em: https://www.cyberark.com/resources/threat-research-blog/fantastic-rootkits-and-where-to-find-them-part-3-arm-edition. Acesso em: 13 jun. 2024.

SHAID; MAROOF. **In memory detection of Windows API call hooking technique** [2015]. Disponível em: https://www.researchgate.net/publication/283480063\_In\_memory\_detection\_of\_Windows\_API\_call\_hooking\_technique. Acesso em: 13 jun. 2024.

SIKANDER. **AV/EDR Evasion Using Direct System Calls (User-Mode vs kernel-Mode)** [2022]. Disponível em: https://medium.com/@merasor07/av-edr-evasion-using-direct-system-calls-user-mode-vs-kernel-mode-fad2fdfed01a. Acesso em: 13 jun. 2024.

TANDA. **DdiMon** [2016]. Disponível em: https://github.com/tandasat/DdiMon. Acesso em: 13 jun. 2024.

TECHTARGET. **Second-level address translation (SLAT).**[2014]. Disponível em: https://www.techtarget.com/whatis/definition/second-level-address-translation-SLAT. Acesso em: 13 jun. 2024.

TOLHETI. **Learn about hypervisors, system virtualization and how it works in a cloud environment.**[2011]. Disponível em: https://developer.ibm.com/articles/cl-hypervisorcompare. Acesso em: 13 jun. 2024.

WANG. **Syscall-Monitor.**[2017]. Disponível em: https://github.com/hzqst/Syscall-Monitor. Acesso em: 13 jun. 2024.