UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia

Departamento de Computação
Bacharelado em Engenharia de Computação

Víctor Cora Colombo

Desenvolvimento de um mecanismo de proteção contra a instalação de módulos maliciosos no Kernel Linux baseado em *syscall hooks*

São Carlos - SP

2022

Víctor Cora Colombo

Desenvolvimento de um mecanismo de proteção contra a instalação de módulos maliciosos no Kernel Linux baseado em *syscall hooks*

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Engenharia de Computação da Universidade Federal de São Carlos como requisito parcial para a obtenção do título de Bacharel em Engenharia de Computação.

Orientação Prof. Dr. Hélio Crestana Guardia

São Carlos - SP

2022

“Talk is cheap. Show me the code.“
(Linus Torvalds)

Resumo

Apesar de categorizado como de arquitetura monolítica, o kernel Linux possui a capacidade de estender sua funcionalidade por meio da instalação de módulos, chamados de LKM. Devido à sua importância, o Linux torna-se alvo prioritário de hackers mal-intencionados que buscam realizar ataques de alto impacto contra sistemas computacionais com diversos objetivos. A simplicidade do processo de adição de código de forma modular chama a atenção desses hackers, que buscam instalar programas conhecidos como *rootkits* de kernel com o objetivo de infectar o sistema operacional.

O presente trabalho tem como objetivo propor uma nova forma impedir a infecção de um sistema Linux por *rootkits*, sem a necessidade de recompilar o kernel ou reiniciar o sistema computacional. Isto é feito por meio do uso de assinaturas de módulos e chaves criptográficas. O resultado desta proposta é um módulo que, quando carregado ao sistema, deve impedir a instalação de outros módulos que não tenham sido assinados pelo verdadeiro administrador desse. Esse módulo e ferramentas auxiliares a serem desenvolvidos ajudarão a realizar a assinatura e verificá-las quando a instalação é executada.

Palavras-chave: Linux kernel, segurança, rootkit.

Abstract

Even though the Linux kernel is categorized as being monolithic, it has the capability of extending its functionalities by adding code through modules that are added during runtime. Due to its importance, Linux is a preferential target for a high number of ill-intentioned hackers seeking to perform high impact attacks targeting these computer systems. The feature of easily adding code to the kernel using modules draws the attention of these hackers, whom seek to install programs known as kernel rootkits aiming to infect the operating system.

This work will propose a new way to prevent rootkits from infecting a Linux system, without the need to recompile the kernel or reboot the machine. This is done through the use of module signing and cryptographic keys. The result will be a module that, when loaded into the system, should prevent the installation of other modules that have not been signed by the real system administrator. This module and other auxiliary tools will help to sign and verify them when the installation is performed.

Keywords: Linux kernel, security, rootkit.

Lista de ilustrações

Lista de Códigos

Sumário

1 Introdução

Sistemas operacionais são o componente de software responsável pelo gerenciamento dos recursos de hardware, além de prover aos programas de usuário uma visão mais simples e abstrata da máquina (TANENBAUM, 2009) por meio da implementação de uma interface de comunicação entre os componentes físicos e os programas que nele são executados. Computadores possuem, primariamente, dois tipos de modo de operação: de kernel (conhecido como supervisor), e de usuário. Quando executado no primeiro modo, o código possui acesso completo ao hardware, com permissão irrestrita de manipulação da memória e de comunicação por meio de I/O.

O Linux, um dos principais sistemas operacionais da atualidade, encontra-se presente em diversos tipos de máquinas, desde sistemas embarcados, computadores pessoais, servidores web, até infraestrutura crítica como serviços de *firewall*, DNS, entre outros. Apesar de ser classificado como um kernel de arquitetura monolítica, o Linux apresenta a funcionalidade de inserção de código de forma modular em tempo de execução do sistema, estendendo suas capacidades de acordo com a necessidade (CORBET; RUBINI; KROAH-HARTMAN, 2005). Os *Linux Kernel Modules* (LKMs), como são conhecidos, permitem a adição de funcionalidades que passam a ser executadas como parte do kernel, com os mesmos privilégios irrestritos do código originalmente carregado durante a inicialização da máquina. Sua instalação é feita de forma simples, sendo necessário apenas a execução de uma chamada de sistema realizada por um usuário privilegiado.

Devido à sua importância no cenário dos sistemas operacionais, o Linux torna-se alvo de destaque por hackers mal-intencionados que buscam realizar ataques de alto impacto contra sistemas computacionais com diversos objetivos. A simplicidade do processo de adição de código de forma modular chama a atenção desses hackers, que buscam instalar programas conhecidos como *rootkits* de kernel com o objetivo de infectar o sistema operacional, tomando controle sobre ele de forma eficiente e deixando a menor quantidade de rastros possíveis. Dessa maneira, é interessante que haja uma forma de impedir que módulos maliciosos sejam instalados sem conhecimento do verdadeiro administrador do sistema, ao mesmo tempo que módulos benignos ainda possam ser adicionados. Atualmente, o Linux possui diversos mecanismos que podem ser utilizados com esse objetivo, porém com desvantagens que incluem desde complicadas configurações até a necessidade de recompilação do kernel ou acesso a BIOS da máquina, o que nem sempre é possível ou pode acarretar *downtimes* indesejados.

Este projeto propõe a implementação de um mecanismo de defesa com o objetivo de permitir a instalação de novos módulos no kernel Linux apenas pelo indivíduo que realmente administra o sistema, em situações que outros mecanismos semelhantes não podem ser aplicados de forma simples, enquanto torna impossível a um usuário malicioso com permissões de *root* realize essa ação. Esse mecanismo é implementado na forma de um módulo de kernel, utilizando-se da técnica de substituição de entradas na tabela de *system calls* do sistema operacional para modificar as rotinas de instalação de módulos, adicionando checagens antes da instalação desses.

1.1 Objetivos

1.1.1 Objetivo Geral

O presente trabalho tem como objetivo propor, apresentar e detalhar um método para permitir apenas ao verdadeiro administrador do sistema a instalação de LKMs, sem a necessidade de recompilar o kernel ou acessar a interface EFI do sistema para adicionar chaves criptográficas. O resultado esperado será um módulo de kernel Linux, que terá a função de impedir a instalação de outros módulos se não pelo verdadeiro administrador do sistema. Esse módulo e ferramentas auxiliares desenvolvidos no decorrer do trabalho ajudarão a assinar módulos e verificá-los quando sua instalação é solicitada a partir das chamadas de sistema relevantes.

1.1.2 Objetivos Específicos

Os objetivos específicos abaixo apresentam o caminho a ser seguido para a implementação de todas as funcionalidades esperadas da proteção contra a instalação de módulos indesejados no Kernel do Linux:

- Estudo de como binários e mensagens são assinados e verificados em outros contextos na computação.
- Identificação de formas de assinar módulos de Kernel, que são basicamente arquivos do formato *Executable and Linkable Format* (ELF).
- Desenvolvimento de ferramentas para introduzir assinaturas ao binário do LKM.
- Desenvolvimento de um módulo de kernel para verificar assinaturas de binários. Esse módulo deve ser capaz de garantir que qualquer forma de instalação de LKMs passe por ele. Deverá ser feito um estudo de formas que módulos podem ser instalados e como desabilitá-las sem que o usuário *root* seja capaz de reabilitá-las.
- Documentar brevemente o uso da API criptográfica do Linux para cálculo de hashes e verificação de assinaturas dentro do código do kernel.

1.2 Organização do Trabalho

O restante desta monografia está organizado em mais 5 capítulos, sendo eles:

- Fundamentação Teórica, no Capítulo 2, em que diversos conceitos relacionado ao trabalho serão abordados. Inclui-se explicações relacionadas aos conceitos de sistemas operacionais, kernel, Linux, chamadas de sistema, módulos no kernel Linux, *rootkits*, arquivos ELF, assinaturas digitais, entre outros.
- Módulo de Defesa, no Capítulo 3, onde o tópico principal do trabalho será apresentado e discorrido, com a especificação do mecanismo de proteção proposto.

- Metodologia, no Capítulo 4, onde será comentado sobre o ambiente e os testes realizados para verificar a implementação da proteção.
- Resultados, no Capítulo 5, em que os testes executados são apresentados e discutidos.
- Conclusão, no Capítulo 6, onde vê-se uma breve análise das contribuições apresentadas e de trabalhos futuros.

2 Fundamentação Teórica

2.1 Linux Kernel

O Linux é um kernel de sistema operacional de código aberto criado em 1991 por Linus Torvalds (WELSH; DALHEIMER; KAUFMAN, 1999), e posteriormente adotado como kernel do projeto GNU[1]. Seu kernel é dito *Unix-like*, ou seja, apesar de não incluir aplicações e ferramentas de sistemas *Unix* (BOVET; CESATI, 2005), implementa sua API da mesma forma, definida pelos padrões *POSIX* (LOVE, 2010).

O objetivo de sistemas operacionais, e portanto do Linux, é permitir e organizar a interação entre programas e o hardware, ao mesmo tempo que disponibiliza um ambiente estável para a execução desses programas. Isso é feito a partir de seus diversos componentes, tais como:

- **Gerenciamento de processos**: criação de destruição de processos; comunicação entre processos (IPC), sinais e pipes; *scheduler*.
- **Gerenciamento de memória**: Alocação e organização da memória virtual e física do sistema, incluindo o controle da interação entre processos e a memória.
- **Sistemas de arquivos**: Gerenciamento de arquivos; utilização de diversos tipos de discos e sistemas de arquivos no sistema ao mesmo tempo; interação entre processos e os sistemas de arquivos.
- **Gerenciamento de dispositivos**: Controle de dispositivos, garantindo a interação organizada entre software e hardware. Normalmente feito com o uso de *device drivers*.
- **Gerenciamento de rede**: Envio e recebimento de pacotes; implementação de diversos protocolos no kernel; passagem de pacotes recebidos para os processos no espaço de usuário.

2.1.1 Arquitetura do kernel

Por sua categoria, o Linux é considerado monolítico, significando que seu kernel é compilado em um grande binário, sendo o sistema operacional executado completamente como um único programa em modo privilegiado (TANENBAUM, 2009). No modelo monolítico, quando uma parte do código deseja interagir com outra realiza-se a importação do método desejado seguido de sua invocação direta, tendo a vantagem de ser extremamente simples de programar e utilizar, além de proporcionar bom desempenho, um dos principais objetivos de um kernel. Por outro lado, uma de suas desvantagens é o fato de todo o sistema estar acoplado como parte de um único processo, onde implica-se que uma única falha pode causar um erro fatal em todo o kernel, ou que qualquer falha de segurança pode causar o comprometimento de partes críticas do sistema.

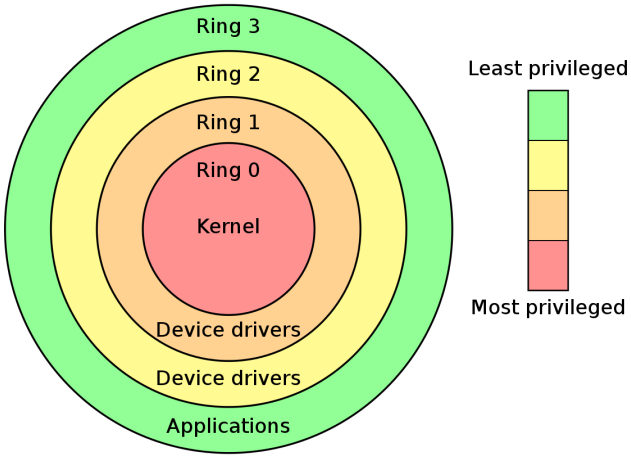
[1] <https://www.gnu.org/home.en.html>

Essa organização se opõe a outros modelos, como o *microkernel*, em que diferentes funcionalidades do código são executadas de forma separada e se comunicam por meio do mecanismo de passagem de mensagens. No caso do Linux, a opção pela utilização do kernel monolítico se deve à questão de desempenho, dado que não há a necessidade de troca de contexto ou preparação e passagem de mensagens quando métodos são chamados (SILBERSCHATZ; GALVIN; GAGNE, 1999).

2.1.2 Modos de execução

Sistemas operacionais utilizam dois modos principais de execução: de usuário e de kernel. No espaço de usuário (Ring 3 em x86), os processos possuem privilégios reduzidos, sendo que muitas instruções e ações não podem ser executadas, tais como acesso direto ao hardware, sendo necessário a comunicação com o sistema operacional via chamadas de sistema para realizar tarefas como leitura de disco e memória, por exemplo. Isso acontece mesmo quando o usuário é o *root*. Já em espaço de kernel (Ring 0 em x86), o código tem privilégios totais e acesso a funcionalidades do hardware. A Figura 1 mostra o modelo teórico da hierarquia de proteção dos modos de execução, representada normalmente na forma de anéis. Na prática, o Linux utiliza apenas o Ring 0 para o kernel e drivers, e Ring 3 para aplicações de usuário.

Figura 1 – Representação dos modos de execução na arquitetura x86



Fonte: Wikimedia Commons.

O kernel do Linux roda por si só, não fazendo uso de quaisquer bibliotecas de espaço de usuário, tais como a *GNU libc* (LOVE, 2010). Dessa forma, diversas funções disponíveis para serem usadas em códigos de programas em nível de usuário não estão presentes para uso no kernel, enquanto outras são reimplementadas, como é o caso do *printk* (semelhante ao *printf*, mas não idêntico), *memcpy*, entre outros. Pertinente a esse trabalho, isso significa que não é possível utilizar as bibliotecas criptográficas consagradas na linguagem C, como `openssl` por exemplo, fazendo-se necessário o uso da API provida pelo próprio kernel.

2.1.3 Syscalls

De forma a permitir que processos em nível de usuário possam acessar recursos aos quais apenas o kernel possui os privilégios necessários, sistemas operacionais definem uma interface conhecida como chamadas de sistema (*syscalls*). Essa permite que o acesso a recursos como rede, hardware, memória, etc. seja feito de forma controlada e segura, com controle de permissões, gerenciamento de recursos, garantia de estabilidade, entre outros (LOVE, 2010), já que todas as requisições passam a ser processadas internamente pelo kernel, que faz seu controle e garante que as permissões de acesso são verificadas quando necessário.

Apesar de haver semelhanças, cada arquitetura de processadores possui sua própria lista de chamadas de sistemas. Por exemplo, na arquitetura x86-64, o arquivo `unistd.h`[2] define as chamadas disponíveis e sua numeração, que é um valor único utilizado pelas aplicações para executá-las, por meio da função *syscall*[3]. Dessa forma, códigos tanto em espaço de usuário quanto de kernel se tornam acoplados à arquitetura caso dependam diretamente do número ou dos parâmetros de uma chamada de sistema.

2.1.3.1 Inicialização da system call table

O kernel do Linux possui uma tabela chamada *system call table*, representada pela array `sys_call_table` definida no código do arquivo `syscall_64.c`[4], que armazena em cada uma de suas posições a referência para a função do kernel correspondente à chamada de sistema de número equivalente. Inicialmente, todas as entradas da array são inicializadas com o valor `sys_ni_syscall`[5], que indica uma chamada de sistema não implementada (0XAX, 2020). Essa função apenas retorna `-ENOSYS`, indicando que a chamada de sistema não existe ou não foi implementada. Portanto, caso um programa faça uma chamada de sistema não implementada para aquela arquitetura, tal erro será retornado por padrão.

A inicialização da *array* com os valores corretos acontece imediatamente em seguida, com a inclusão do header `asm/syscall_64.h`. Dessa forma, todas as entradas correspondentes a chamadas de sistema existentes na arquitetura passam a apontar para os *handlers* que implementam as chamadas de sistema específicas, com nome `sys_[nome_da_syscall]` (por exemplo `sys_read` no índice 0, ou `sys_finit_module` no índice 273). O protótipo dessas funções pode ser encontrado no arquivo `include/linux/syscalls.h`[6], enquanto suas implementações estão espalhadas pelo código, na forma da macro `SYSCALL_DEFINE`.

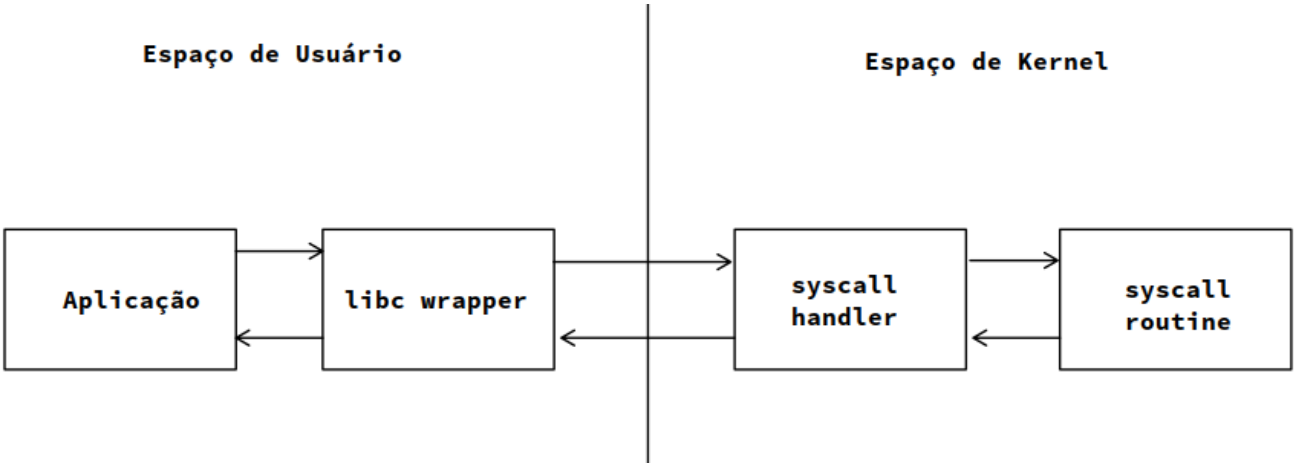
2.1.3.2 Caminho de uma chamada de sistema

Apesar de poder ser realizada de forma direta a partir do uso da função *syscall*, a execução de uma chamada de sistema por uma aplicação em espaço de usuário normalmente não o faz dessa maneira. No seu lugar, as chamadas são feitas por meio de funções expostas pela biblioteca padrão do

[2] <https://elixir.bootlin.com/linux/v5.19/source/tools/include/uapi/asm-generic/unistd.h>
[3] <https://man7.org/linux/man-pages/man2/syscall.2.html>
[4] <https://elixir.bootlin.com/linux/v5.19/source/arch/x86/entry/syscall_64.c#L16>
[5] <https://elixir.bootlin.com/linux/v5.19/source/kernel/sys_ni.c#L20>
[6] <https://elixir.bootlin.com/linux/v5.19/source/include/linux/syscalls.h>

C conhecidas como *wrapper routines*, cujo único objetivo é preparar os argumentos, colocando-os nos registradores específicos, e realizar a chamada para o kernel (FREE SOFTWARE FOUNDATION, 2021c).

Figura 2 – Representação simplificada do caminho de uma execução de chamada de sistema, passando pelo *wrapper* na *libc*.



Fonte: De autoria própria.

Na arquitetura x86-32, quando a instrução `int 0x80` é executada por um processo em espaço de usuário, uma interrupção síncrona de software é gerada, fazendo com que a CPU procure em sua *Interrupt Descriptor Table* (IDT) o código que deve ser executado no caso dessa interrupção, conhecido como *interrupt handler*. É feita então a troca de contexto para dentro do espaço do kernel onde, no Linux, a função correspondente ao *system call handler* é chamada. Antigamente, essa função era chamada `system_call()`, porém nas versões mais modernas do kernel ocorreu sua substituição pelo método `entry_INT80_32`, responsável por construir uma `struct pt_regs` com os registradores com os quais a chamada de sistema foi executada e então invocar `do_int80_syscall_32()`.

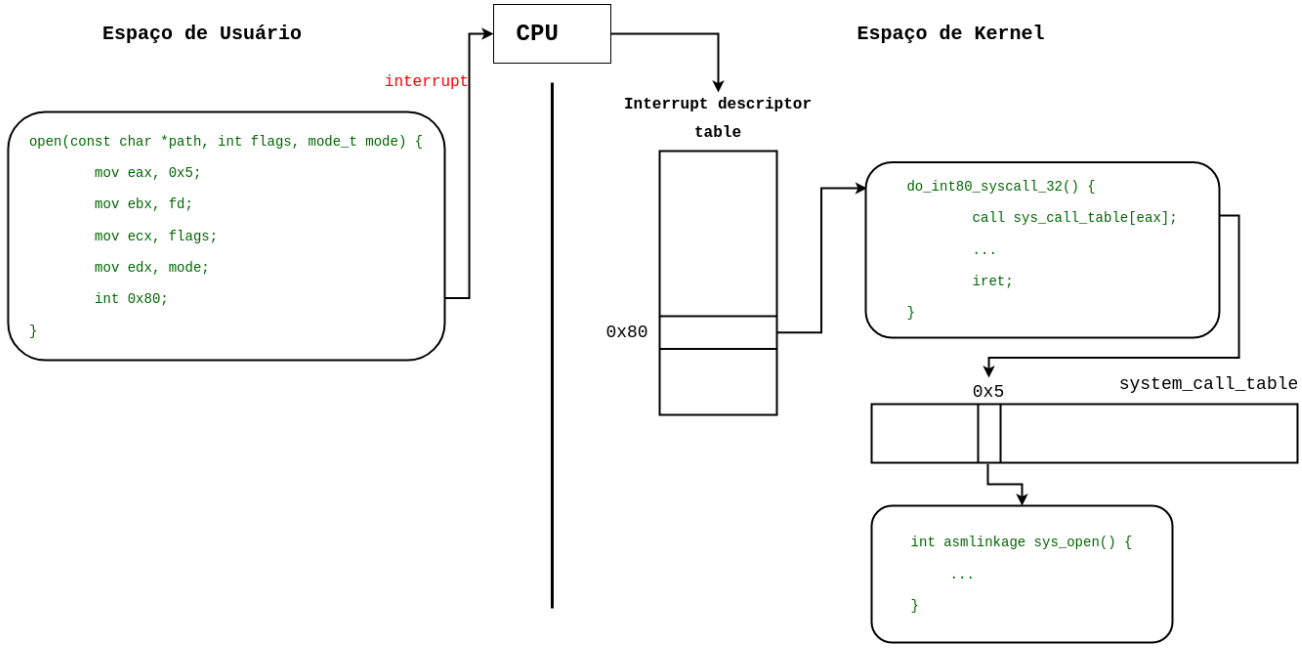
Nesse ponto, a array `sys_call_table` é desreferenciada de forma a realizar a chamada da função que executa a chamada definida pelo valor presente no registrador `eax`. A Figura 3 mostra a representação de um exemplo desse processo para a *syscall* `open`.

Já as CPUs da arquitetura x86-64 implementam a instrução *syscall*, que permite saltar de forma mais eficiente do espaço de usuário para dentro do kernel, sem passar pelo mecanismo de interrupções, que é mais lento. Quando executada, essa instrução invoca o *handler* de *syscalls* do kernel com privilégios de *ring* 0, a partir do carregamento do endereço desse *handler* via registradores `MSR` presentes na CPU[7].

Esse *handler* realiza o salto para o código presente em `entry_64.S`, que em seguida é responsável por construir uma `struct pt_regs` e chamar o método `do_syscall_64()`[8], passando como parâmetro a estrutura construída previamente e o número da *syscall*, obtido a partir do registrador `rax`. Por sua vez, invoca-se `do_syscall_x64`[9]. Nesse ponto, verifica-se a validade do número da chamada de sistema, e a função referenciada correspondente a esse número na tabela

[7] <https://www.felixcloutier.com/x86/syscall>
[8] <https://elixir.bootlin.com/linux/v5.19/source/arch/x86/entry/common.c#L73>
[9] <https://elixir.bootlin.com/linux/v5.19/source/arch/x86/entry/common.c#L40>

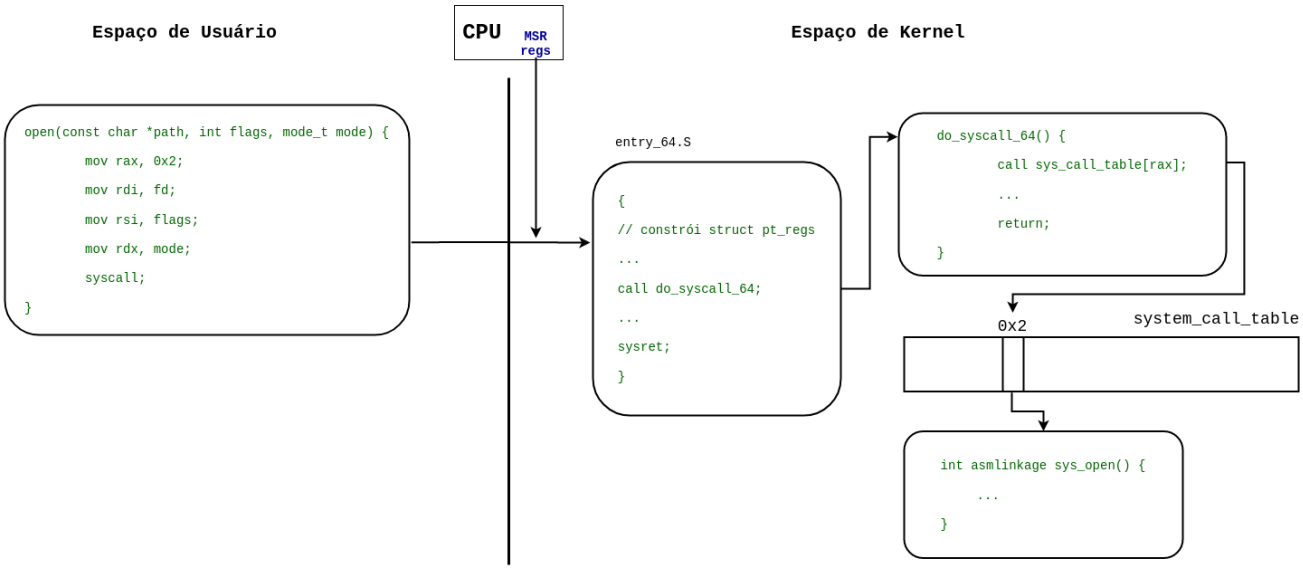
Figura 3 – Representação detalhada do caminho de uma execução de chamada de sistema em x86-32.



Fonte: De autoria própria.

de *syscalls* `sys_call_table` é executada. Sendo assim, após o ingresso no espaço do kernel, o funcionamento é muito semelhante entre as arquiteturas, mudando apenas as funções especializadas que são chamadas no caminho.

Figura 4 – Representação detalhada do caminho de uma execução de chamada de sistema em x86-64.



Fonte: De autoria própria.

2.1.3.3 Resultado de uma chamada de sistema

No fim de sua execução, a *syscall* retorna para o handler em `entry_64.S`, salvando o resultado da chamada no registrador `rax`, além de restaurar o contexto ao estado original antes de regressar ao espaço de usuário. Chamadas de sistema também retornam resultados, cujo valor é 0 em caso de

sucesso, e -1 em caso de alguma falha, sendo que nessa situação o código do erro ocorrido é salvo na variável `errno`, de forma a ser conferida pelo programa em espaço de usuário. Esse código é um valor positivo, e seus vários valores possíveis podem ser conferidos na documentação do kernel[10]. Valores interessantes e comum de ocorrerem incluem `EPERM Operation not permitted`, `EACCES Permission denied`, e `EINVAL Invalid argument`.

Já no kernel, o comportamento é um pouco diferente do que é visto em espaço de usuário. *Syscalls* retornam valores negativos em caso de erro (por exemplo `return -EPERM`), sendo o *wrapper* da `glibc` então responsável por fazer as modificações necessárias (ou seja, retornar -1 e colocar o resultado em `errno`).

2.1.4 Programação Orientada a Objetos no Kernel

Programação orientada a objetos (POO) é um paradigma em que o software é modelado como uma coleção de abstrações, de forma que cada uma dessas é chamada de classe, cujas instâncias recebem o nome de objeto, e procuram representar um aspecto do sistema (MDN WEB DOCS, 2022). O programador define e implementa uma interface pública pela qual outros componentes do código podem utilizar para se comunicar com o objeto, enquanto mantém seus dados internos (estado) escondidos, visíveis apenas para si mesmo. Isso permite que o programador exponha uma API concisa e coesa aos usuários, gerenciando as dificuldades relacionados a estrutura interna do conceito sendo modelado.

O paradigma normalmente é implementado utilizando mecanismos embutidos nas linguagens de programação, como classes, heranças, interfaces, entre outros. Tais funcionalidades estão presentes nas chamadas linguagens orientadas a objeto, como é o caso do C++ e Java. Porém, esses mecanismos são apenas ferramentas de implementação, não sendo necessários para a criação de um software que utilize dos conceitos de POO. Desse modo, mesmo escrito quase inteiramente com o uso da linguagem C, o kernel do Linux utiliza extensivamente o conceito de POO no seu código de forma a providenciar extensibilidade e reusabilidade. Para isso, utiliza-se de *structs* para guardar o estado do "objeto", e de *non-member functions*, ou seja, funções que não são membro de classes, e sim que recebem um objeto (nesse caso, uma *struct*) como um de seus argumentos.

2.2 Linux Kernel Modules (LKMs)

Apesar de monolítico, o kernel do Linux permite a adição de funcionalidades durante sua execução por meio de módulos conhecidos como *Linux Kernel Modules* (LKMs). Módulos são binários que não formam um executável por si só, e que são dinamicamente vinculados (*linked*) com o kernel durante sua execução (CORBET; RUBINI; KROAH-HARTMAN, 2005). O kernel não precisa ter conhecimento algum de quais módulos serão carregados no futuro, sendo ambos independentes um do outro (SILBERSCHATZ; GALVIN; GAGNE, 1999).

Essa *linkagem* é feita com os programas `insmod` ou `modprobe`. Sua diferença consiste basicamente no fato do primeiro apenas instalar o módulo solicitado, enquanto o segundo é capaz de

[10] <https://man7.org/linux/man-pages/man3/errno.3.html>

encontrar as dependências necessárias por aquele módulo e carregá-las na ordem correta. Por exemplo, o comando `modprobe vhost_net` resulta no carregamento no sistema não só do módulo `vhost_net`, mas também de suas dependências: `vhost`, `vhost_iotlb`, `tun`, e `tap`. `modprobe` atinge esse comportamento por meio da leitura do arquivo `modules.dep`, gerado durante o processo compilação dos módulos. Um módulo é considerado dependente de outro caso ele utilize um símbolo (variáveis ou funções) que o outro define e exporta (SALZMAN; BURIAN; POMERANTZ, 2007).

Da mesma maneira, a remoção de módulos pode ser feita com os comandos `rmmod` ou `modprobe` (por meio do parâmetro `-r`). Assim como no processo de instalação, `modprobe` se difere no fato de que ele também remove suas dependências, quando essas não são utilizadas por mais nenhum outro módulo carregado. É também possível obter informações sobre os módulos que estão carregados no sistema no momento, por meio da ferramenta `lsmod`, que obtém suas informações a partir da leitura do arquivo `/proc/modules`.

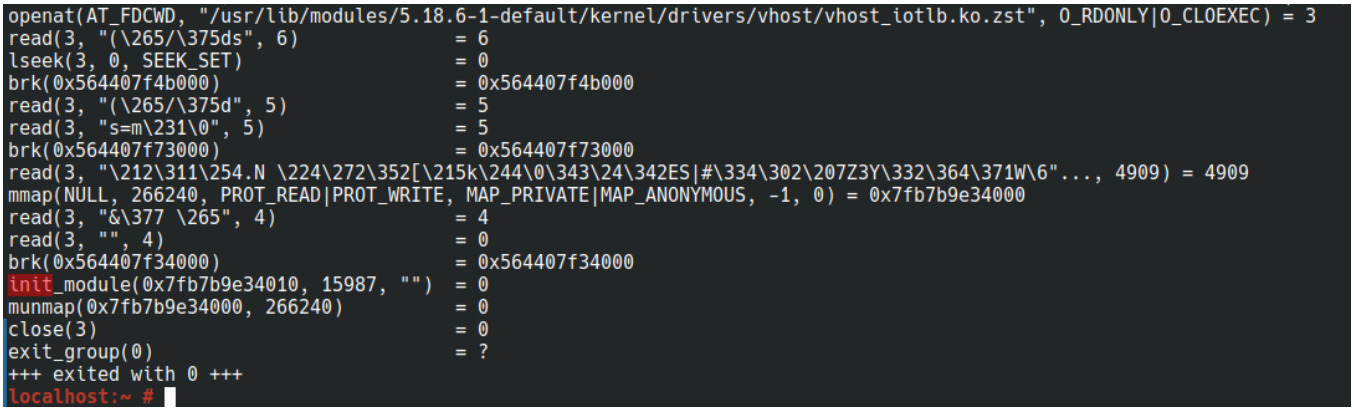
Cada módulo é compilado exclusivamente para uma versão específica do kernel Linux. Portanto, quando compilado um módulo terá seu carregamento rejeitado em qualquer outro kernel que não o dessa versão específica. Essas checagens são adicionadas no momento da compilação do módulo e verificadas quando o mesmo é carregado. De acordo com a documentação do Linux (FREE SOFTWARE FOUNDATION, 2021b), a verificação inclui uma *string* contendo a versão do kernel e uma descrição de seus atributos (por exemplo, a arquitetura e tipo de CPU). Além disso, uma hash de cada símbolo que o kernel utiliza também é salva no binário para verificação durante seu carregamento.

O carregamento de LKMs acontece por meio das chamadas de sistema `init_module` e `finit_module`. Primeiramente, a chamada de sistema `init_module` (ou `finit_module`) é feita para notificar o kernel de que um módulo deve ser carregado. O controle então é passado para o núcleo do sistema operacional, onde a função `sys_init_module()` é invocada, realizando ações como verificação de permissão, cópia do *buffer* do usuário para o espaço de memória do kernel, validação do arquivo ELF, resolução de símbolos, e outras etapas necessárias para o carregamento do módulo por completo (KUMAR, 2013). Só então a chamada à função `module_init`, declarada no código do módulo como ponto de entrada de sua execução, é efetuada. O objetivo da *syscall* `init_module` é carregar uma imagem de um binário ELF previamente alocado na memória (utilizando-se das chamadas `open` e `read`) para o espaço do kernel. Esse comportamento pode ser visto na Figura 5, em que o uso de `insmod` nesse sistema causa a abertura do arquivo do módulo, seguido do seu carregamento para a memória por meio da *syscall* `open`, e finalmente chamando `init_module` com o endereço de memória que a imagem do ELF do módulo está carregada.

Já `finit_module` se diferencia pelo fato de carregar o módulo a partir de um *file descriptor* (FREE SOFTWARE FOUNDATION, 2021b). Isso pode ser visto na Figura 6, onde o binário é aberto, e seu descritor de arquivo é passado como argumento para a chamada de sistema. A escolha pelo uso de cada depende da implementação do programa `insmod` no sistema, ou do programador que faz a chamada diretamente se esse for o caso.

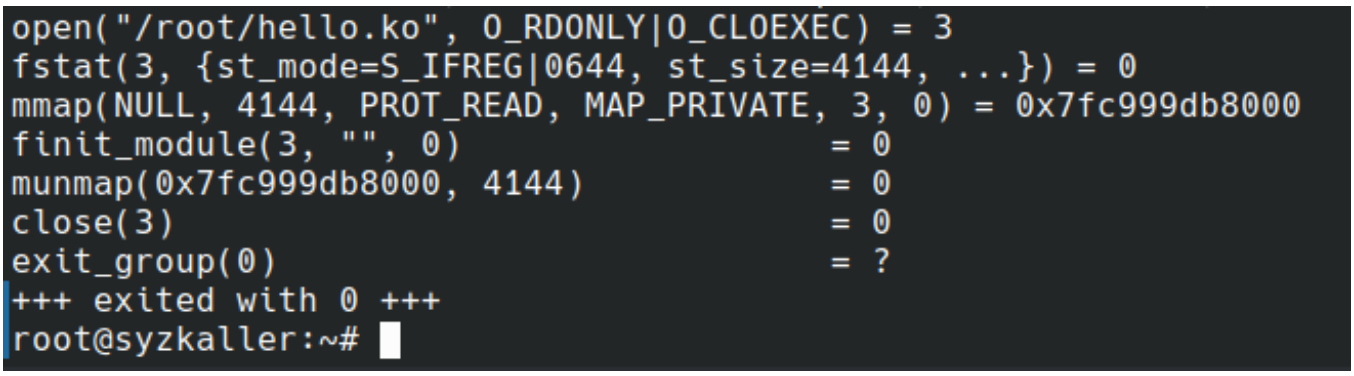
Por fim, a remoção de módulos acontece por meio da *syscall* `delete_module`. Essa chamada recebe como parâmetro o nome do módulo, executando sua remoção do espaço do kernel por meio

Figura 5 – Resultado do comando `strace insmod` mostrando as chamadas de sistema que são feitas durante o carregamento de um módulo em que a chamada utilizada é `init_module`.



Fonte: De autoria própria.

Figura 6 – Resultado do comando `strace insmod` mostrando as chamadas de sistema que são feitas durante o carregamento de um módulo em que a chamada utilizada é `finit_module`.



Fonte: De autoria própria.

da chamada da função registrada via `module_exit` no seu código-fonte.

As funções da API de módulos do kernel Linux, `module_init` e `module_exit`, são usadas para registrar os pontos de entrada e saída do código do módulo. Ou seja, quando o módulo é carregado, `module_init` é chamado e deve ser utilizada pelo desenvolvedor para fazer sua inicialização, com alocação de memória e preparação de recursos necessários para o funcionamento adequado do mesmo. Já `module_exit` é chamado antes da remoção total do módulo, sendo utilizada para limpar os recursos obtidos previamente durante sua inicialização ou execução.

2.3 Linux Security Modules (LSMs)

LSM é uma framework que provê ao kernel suporte a ”módulos de segurança”, incluindo a implementação de políticas de controle de acesso, *sandboxing*, entre outros (FREE SOFTWARE FOUNDATION, 2022). Ela funciona por meio do uso de *hooks* em partes críticas do kernel que são de interesse relativo a segurança, como o momento da abertura de arquivos, por exemplo. Esses *hooks* já são definidos de forma prévia no código do Linux, sendo que para adicionar novos é necessário

modificar o código do kernel e enviar o *patch* para inclusão no *upstream*[11].

Uma lista de *hooks* disponíveis para uso por LSMs se encontra disponível para consulta na documentação do kernel[12]. No contexto de carregamento de LKMs, o *hook* `kernel_module_request` é chamado quando um módulo está sendo carregado. Contudo, não há *hooks* disponíveis para a verificação de remoção de módulos, de forma que uma solução baseada em LSMs para o problema proposto nesse trabalho teria de incluir a adição de código ao kernel, e portanto não seria imediatamente retro-compatível com versões antigas desse.

Apesar do nome, os *Linux Security Modules* não são módulos de verdade, no sentindo de que não é possível carregá-los de forma dinâmica ao kernel. Na realidade eles são adicionados estaticamente no momento da compilação deste. Dessa forma, para adicionar ou modificar LSMs, é necessário recompilar o kernel por completo. Exemplos do uso dessa *framework* incluem o *Apparmor*, utilizado no Ubuntu, *SELinux*, utilizado no Fedora, e até as *capabilities* do Linux.

2.4 Rootkits

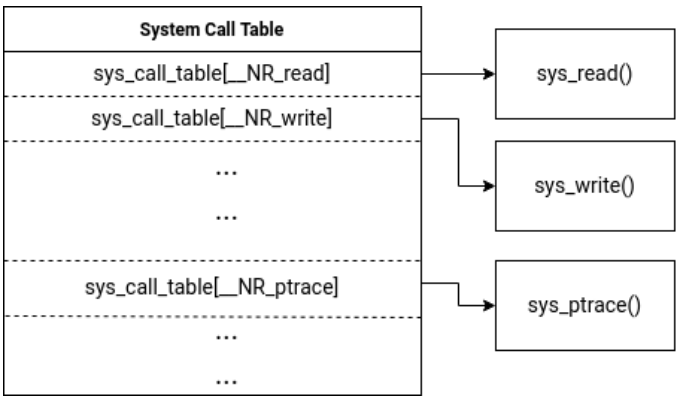
Rootkits são programas costumeiramente maliciosos que funcionam a partir da alteração do comportamento do sistema, interferindo na comunicação do sistema operacional consigo mesmo e com as aplicações do sistema (SPARKS; BUTLER, 2005). Seu objetivo principal é omitir o fato de uma invasão ter ocorrido no sistema, de forma a prolongar a persistência no mesmo. O termo *rootkits* significa um conjunto de ferramentas para obter e manter acesso de administrador (*root*) ao sistema de forma indetectável e persistente (HOGLUND; BUTLER, 2006). Por si só, *rootkits* não são maliciosos, sendo a definição comparável ao de um *backdoor*, que pode ser usado de forma legítima para atividades legais e úteis, como acesso remoto ao sistema para fins de suporte, controle de máquinas dos funcionários pelo departamento de TI de uma empresa, ou parte de uma ação de investigação policial.

Em relação à sua categorização, *rootkits* são divididos em diversos tipos tais como de hardware, de memória, de aplicativos, entre outros. Para o presente trabalho, o tipo mais importante são os de kernel. Neste tipo, o *rootkit* é instalado dentro do espaço do supervisor, normalmente via a instalação de um módulo, mas também por meio da exploração de vulnerabilidades no sistema ou de funcionalidades de depuração do sistema operacional que foram deixadas ativas e desprotegidas. Um dos principais mecanismos de atuação desse tipo é a técnica de *syscall hijacking*. Nela, a *system call table* é alterada de forma que uma ou mais de suas entradas são substituídas, causando a execução de outro código que não aquele originalmente responsável pelo tratamento da chamada de sistema correspondente. Esse código então atua de forma maliciosa para modificar o resultado da chamada, por exemplo removendo valores do resultado para esconder a presença de arquivos específicos no sistema. Só então o código original da chamada de sistema é executado, de forma que o comportamento ainda é semelhante ao original, fazendo o usuário não notar qualquer diferença em relação a um sistema inalterado. As Figuras 7 e 8 demonstram esse processo.

[11] O termo *upstream* se caracteriza pelo código que foi revisado e aceito pelo projeto original, nesse caso estando presendo na árvore do Linus Torvalds.

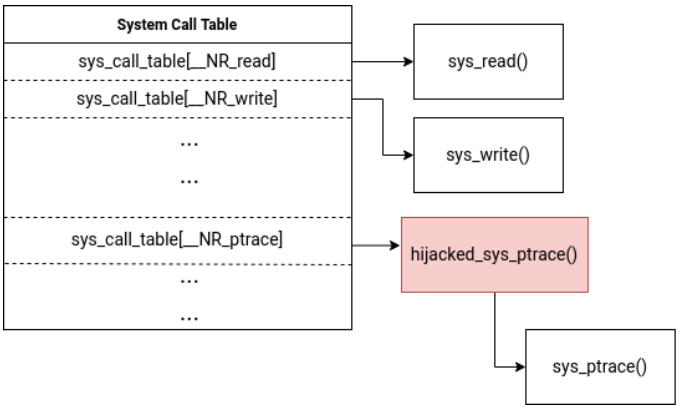
[12] <https://www.kernel.org/doc/html/v5.2/security/LSM.html>

Figura 7 – Representação da *system call table* em seu estado normal.



Fonte: De autoria própria.

Figura 8 – Representação da *system call table* após ser alterada por um *rootkit* de kernel com o mecanismo de *syscall hijacking*.



Fonte: De autoria própria.

2.5 ELFs

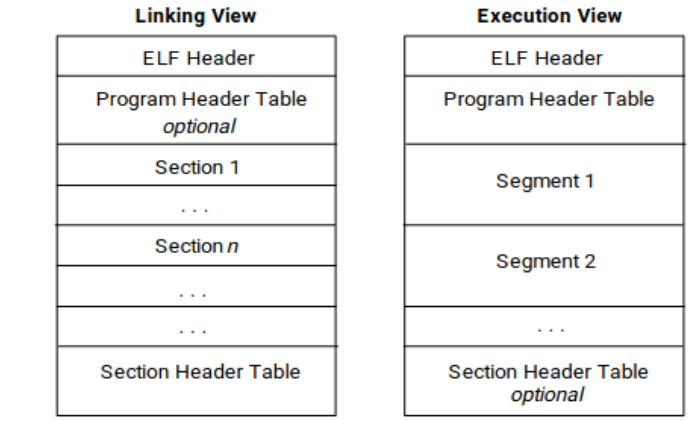
Executable and Linking Format (ELF) é o formato de arquivos binários executáveis no Linux. Esse tipo de arquivo é criado tanto pelo *assembler* quanto pelo *linker* e representa um programa que pode ser tanto utilizado para construir outro executável, ou ser executado diretamente em um processador (TIS COMMITTEE ET AL., 1995). Há três tipos desses arquivos:

- Realocáveis (*relocatable*): código e dados utilizados no processo de *linkagem* com outros objetos, com o objetivo de criar um executável completo.
- Executáveis (*executable*): programa independente que pode ser executado.
- Objeto compartilhado (*shared object*): código e dados que podem sem utilizados para criar outros programas, ou que podem ser carregados pelo *dynamic linker* durante a execução de outro programa.

Arquivos ELF consistem de um *header*, que se encontra sempre no começo do arquivo (*offset* zero), seguidos de uma *program header table* e da *section header table* (FREE SOFTWARE FOUNDATION, 2021a), sendo que esses dois últimos são opcionais, dependendo do contexto. A

Figura 9 mostra como o arquivo se organiza, tanto na situação em que ele deve ser *linkado*, quanto no cenário de um arquivo executável. Essa organização é feita durante o processo de compilação, sendo que a estrutura do ELF no binário é rígida após essa etapa, no sentido de que a posição de início de cada seção é inserida na própria estrutura, significando que para mover ou inserir seções após a compilação seria necessário modificar também as previamente existentes.

Figura 9 – Organização interna de um arquivo ELF.



Fonte: Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, página 1-1.

O cabeçalho (*header*) de um binário ELF está presente sempre nos primeiros bytes do arquivo, e define diversos campos contendo metadados sobre o mesmo, os quais podem ser utilizados por outros programas para verificar a validade do binário, sua arquitetura alvo, características relacionadas ao processo de carregamento na memória, entre outros. Mais especificamente, no processo de instalação de módulos no kernel Linux é esperado que o cabeçalho comece no *offset* 0, caso contrário não há o prosseguimento do mesmo. Esse detalhe será importante no momento de definir onde a assinatura será adicionada ao arquivo. O Código 2.1 apresenta como exemplo a visualização dos cabeçalhos do binário `ping`.

Código 2.1 – Headers do binário ELF `ping`.

```
# readelf -h /usr/bin/ping
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent
     Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x6180
```

```
Start of program headers:          64 (bytes into file)
Start of section headers:          74832 (bytes into file)
Flags:                             0x0
Size of this header:               64 (bytes)
Size of program headers:           56 (bytes)
Number of program headers:         13
Size of section headers:           64 (bytes)
Number of section headers:         30
Section header string table index: 29
```

2.6 Assinatura Digitais

Assinaturas digitais são ferramentas criptográficas que verificam a autenticidade de conteúdos digitais (arquivos, mensagens, etc) por meio da identificação do seu assinador e de que o conteúdo não foi alterado sem permissão (PAUL, 2017). Assinaturas digitais podem confirmar a integridade de uma mensagem (IBM, 2021), providenciando garantias quanto a origem, identidade do criador, momento da assinatura, entre outros.

Elas funcionam por meio do uso da criptografia assimétrica, de chave pública e privada. A chave privada é utilizada para assinar o conteúdo, sendo que o fato dessa ser secreta (em posse apenas do assinador) garante que não seja possível outro participante no processo replicar a assinatura e corromper o sistema. A chave pública, que nesse contexto está contida junto com metadados do seu dono e da assinatura do conteúdo como parte do certificado, é usada para realizar a verificação. Uma chave privada possui uma única chave pública correspondente, e vice-versa, de forma que a confirmação de que a assinatura corresponde àquela chave pública específica garante que essa veio da chave privada que é seu par único.

Na criptografia assimétrica as chaves possuem papeis bem definidos: caso a chave pública seja usada para criptografar, a pública é usada para descriptografar, e vice-versa. O que acontece no processo de assinatura é que o certificado contém ambas a chave pública e a mensagem assinada, sendo a verificação feita a partir do uso dessa chave para descriptografar a assinatura e verificar o conteúdo em relação à hash da mensagem presente.

Contudo, por si só esse esquema não possui qualquer garantia de que a chave pública é proveniente da fonte que alega ser, podendo ter sido substituída por outra de forma a enganar o processo de verificação. Isso pode acontecer por exemplo em um cenário de *supply chain attack*, quando o próprio processo de distribuição de um programa é atacado. Essa garantia é providenciada apenas pelas autoridades de certificação (CA), organizações consideradas seguras e constantemente auditadas que emitem certificados assinados por elas e fazem sua validação.

Os passos da assinatura de um conteúdo são:

1. Primeiramente, calcula-se a hash da mensagem (ou qualquer outro conteúdo, como um arquivo) utilizando algum algoritmo, por exemplo `SHA256`, e criptografa-a com a chave privada, criando

a assinatura.

2. Envia-se o conteúdo para o receptor.

3. Esse, em posse tanto da chave pública do remetente, do conteúdo da mensagem, e da assinatura, primeiro faz a descriptografia com o uso da chave pública. Em seguida, calcula-se a hash da mensagem recebida.

4. É então verificado se ambos os valores são idênticos. Em caso positivo, a assinatura é válida. Caso contrário, o conteúdo não pôde ser verificado, significando que a mensagem recebida é diferente da original ou que não foi assinada com a chave privada correspondente.

O processo inclui a criptografia da hash do conteúdo, e não do conteúdo em si. Não há qualquer impedimento teórico quanto a isso, podendo ser feita a assinatura do conteúdo como um todo, porém esse processo é custoso, sendo suficiente e portanto preferível efetuar o processo sobre a hash, que costuma ser de tamanho muito inferior, e portanto tornando o processo rápido.

Assinaturas não garantem por si só a confidencialidade de uma mensagem, pois o conteúdo em si não é criptografado. O objetivo é apenas verificar sua autenticidade e integridade, além de permitir a não-repudiação.

2.7 Assinatura de LKMs

O kernel Linux já possui a funcionalidade de assinar e verificar módulos (FREE SOFTWARE FOUNDATION, 2018). O parâmetro de configuração de compilação `CONFIG_MODULE_SIG` ativa essa funcionalidade, enquanto `CONFIG_MODULE_SIG_FORCE` impede que módulos não assinados sejam carregados quando a primeira também está ativada. Todavia, a maioria das distribuições Linux não compila seus núcleos com essas configurações habilitadas, permitindo que o usuário *root* instale módulos de forma irrestrita. Dessa maneira, caso o usuário queira embutir a verificação de assinatura de módulos no kernel, ele deve recompilar o mesmo com essa configuração selecionada.

Além da flag `CONFIG_MODULE_SIG_FORCE`, o parâmetro de linha de comando `module.sig_enforce=1` também habilita a verificação de assinaturas no kernel. Toda a checagem é feita dentro do espaço do supervisor, não dependendo de forma alguma de ações no espaço do usuário, o que torna o processo mais seguro contra ataques mais triviais.

Código 2.2 – Resultado da tentativa de carregamento de um módulo quando `module.sig_enforce=1` está ativo.

```
# cat /proc/cmdline
console=ttyS0 root=/dev/sda earlyprintk=serial net.ifnames=0 module.
   sig_enforce=1 nokaslr
# insmod hello.ko
insmod: ERROR: could not insert module hello.ko: Key was rejected by
    service
[ 2726.660460] Loading of unsigned module is rejected
```

Quando a verificação está habilitada, o kernel verifica a assinatura do módulo em relação às chaves públicas disponíveis. Essas chaves são normalmente provenientes da compilação do kernel, que possui um mecanismo para utilizar durante esta etapa um arquivo que contém as chaves, as quais são registradas no sistema e não podem mais ser modificadas, a não ser realizando outra compilação posteriormente.

Além das chaves populadas durante a compilação, é possível adicionar novas chaves por meio da ferramenta `mokutils`. Esse processo requer que a máquina seja reiniciada e configurações sejam feitas no `EFI`, o que nem sempre é possível, por exemplo em cenários de nuvem pública em que normalmente não há acesso direto ao hardware nesse nível. Isso é necessário pois as distribuições não disponibilizam suas chaves privadas para os usuários, sendo necessário adicionar chaves que o dono da máquina criou para serem utilizadas.

Ao utilizar a funcionalidade provida pelo kernel para assinar um LKM, adiciona-se o certificado digital ao final do binário (FREE SOFTWARE FOUNDATION, 2018), de forma que a assinatura estará contida fora da área do ELF propriamente dito, não fazendo parte da estrutura desse. Há então maior facilidade no desenvolvimento e uso da funcionalidade, devido à simplicidade para adicionar e posteriormente encontrar a assinatura no arquivo. Contudo, possui a desvantagem de não ser possível utilizar múltiplas assinaturas para verificar a integridade em etapas diferentes do processo de checagem, dificultando a adição de métodos complementares que utilizem a mesma premissa. Isso contrasta com a outra opção possível de colocação assinatura, que seria adicionar o certificado como parte da estrutura ELF do módulo, cuja desvantagem se apresenta na complexidade de edição do binário original e posterior checagem do valor original, sem o valor embutido.

2.8 Secure boot

`UEFI Secure Boot` é uma tecnologia que visa aumentar a segurança nas fases que antecedem o carregamento do sistema operacional (DEBIAN, 2022). Ela impede que *malwares* possam comprometer o sistema durante essas etapas. Quando a máquina está iniciando, o *firmware* com Secure Boot habilitado checa a assinatura do software que está sendo carregado a cada etapa, como o próprio *firmware*, *bootloader*, sistema operacional, e drivers (MICROSOFT, 2022a).

Para garantir que o código durante o processo de *boot* é executado de forma segura, sem ser modificado, há a necessidade do uso de assinaturas digitais (WILKINS; RICHARDSON, 2013). Essa assinatura está presente em cada seção executável do código que deve ser protegido. Ela então é checada para verificar se o código a ser executado sofreu alterações não permitidas. O modo que isso é feito é que, ao final de cada etapa do processo de inicialização, a assinatura do código da próxima etapa é verificada por meio da comparação com uma chave pública. Como início da cadeia de confiança, o hardware possui uma chave embutida, normalmente assinada pela própria Microsoft. As próximas etapas possuem as chaves das distribuições Linux, do dono da máquina, entre outros, de forma que pessoas com acesso ao hardware podem adicionar novas chaves e portanto permitir a adição de outros códigos no processo, como por exemplo módulos third-party.

O Secure Boot pode ser desabilitado, não sendo parte obrigatória da especificação do UEFI.

Isso normalmente é feito por meio das configurações da BIOS da máquina.

3 Módulo de Defesa

Como forma de impedir a instalação de módulos maliciosos em sistemas Linux, o presente trabalho propõe a implementação de uma Prova de Conceito (PoC) por meio de um LKM, chamado a partir de agora de Módulo de Defesa, que proverá a proteção a partir do momento do seu carregamento no kernel, com o uso de um mecanismo de verificação de assinaturas que serão adicionadas aos binários dos módulos previamente às suas instalações. Pela arquitetura da solução, o verdadeiro administrador do sistema[1] será o único detentor da chave privada a ser usada para assinar os módulos, de forma que qualquer agente malicioso, mesmo com acesso de super-usuário, não terá acesso a ela.

3.1 Chaves pública e privadas e certificado

Para ser propriamente utilizado, o Módulo de Defesa requer uma etapa de configuração prévia. Um dos pilares da segurança provida por esse mecanismo é de que chaves públicas devem podem ser adicionadas apenas pelo dono da máquina. Caso contrário, um agente malicioso poderia adicionar suas próprias chaves e autorizar a instalação de módulos assinados por ele com seu próprio par de chaves. Nessa PoC, a chave pública será parte estática do Módulo de Defesa, devendo ser adicionado manualmente ao código-fonte do mesmo.

Para isso, primeiro é necessário gerar o par de chaves público-privada, sendo que a chave pública será o certificado utilizado para validar a assinatura dos módulos no futuro. A ferramenta `openssl` é utilizada para realizar essa tarefa, com seu comando `genrsa` sendo usado para criar uma chave privada RSA. A escolha pelo uso do algoritmo criptográfico `RSA` se deve aos motivos desse ser um algoritmo considerado seguro e estar disponível para uso na API do kernel. O comando exposto no Código 3.1 é utilizado para criar uma chave `RSA` de 2048 bits.

Código 3.1 – Criação de chave RSA por meio do comando `genrsa`

```
openssl genrsa -out private-key.pem 2048
```

Para fazer a assinatura do binário, foi decidido por adicioná-la ao final do arquivo do módulo, da mesma forma que o kernel do Linux faz em seu próprio mecanismo já existente. Outras opções incluem adicionar ao começo, ou no meio como parte de uma seção do binário ELF. A primeira cria dificuldades desnecessárias pois as chamadas de sistema que cuidam da inicialização dos módulos realizam a verificação de integridade dos mesmo e esperam que esses possuam a estrutura padrão de um arquivo ELF, ou seja, que comecem com o número mágico específico e tenham seus endereços relativos ao começo do arquivo. A outra opção, criar uma seção para a assinatura na estrutura do ELF, aumenta sua complexidade, visto que iria requerer fazer o *parsing* do ELF de forma mais profunda para encontrar a assinatura, além de necessitar remover (ou zerar) essa seção na hora de calcular a assinatura, para deixar o arquivo idêntico ao original. Contudo, a escolha por anexar ao

[1] Não confundir aqui com o usuário *root*. Neste trabalho, a premissa é que o usuário privilegiado possui permissões excessivas, o que tornaria o sistema inseguro no caso de comprometimento de suas credenciais ou na ocasião de uma escalada de privilégios.

final apresenta a característica negativa de conflitar com a implementação já existente, causando conflitos caso ambas venham a ser utilizadas ao mesmo tempo. Considera-se que esse cenário não irá acontecer no trabalho atual.

A assinatura é feita utilizando-se a chave privada gerada anteriormente por meio do comando `openssl dgst` que, de acordo com sua documentação, é usado para gerar assinaturas digitais a partir da hash do arquivo, cujo algoritmo escolhido foi o `SHA256`. O Código 3.2 representa um exemplo de como o módulo `hello.ko` é assinado.

Código 3.2 – Introduzindo assinatura ao módulo `hello.ko`.

```
openssl dgst -sha256 -sign private-key.pem -out hello.ko.sig hello.
   ko
cat hello.ko hello.ko.sig > signed.ko
```

A chave privada pode então ser utilizada pelo `openssl` para realizar a criação do certificado, como mostrado no Código 3.3. Esse certificado está no formato `DER`, o tipo de arquivo de certificados digitais em formato binário, o que é necessário devido a API criptográfica do kernel do Linux apenas aceitar chaves públicas codificadas nesse formato.

Código 3.3 – Comando para realizar a criação do certificado.

```
openssl req -new -x509 -key ./private-key.pem -outform DER -out ./
   cert.der -nodes -days 3650 -subj "/CN=tcc-kernel-module"
```

O certificado gerado ainda não está no formato esperado pelo kernel. Nesse momento o arquivo ainda possui informações extras, como os metadados que compõe o seu cabeçalho. Esses bytes são desnecessários e precisam ser removidos, caso contrário o kernel irá rejeitar a chave e retornar o erro *EINVAL*.

O manual do comando *openssl-asn1parse*[2] explica brevemente as partes que compõe um certificado. Mais especificamente, que a localização da chave pública em si dentro do arquivo DER pode ser encontrada com essa ferramenta, procurando pelo segmento *BIT STRING* no resultado do comando. A Figura 10 mostra que o arquivo analisado possui sua chave começando no *offset* 161, com o cabeçalho do segmento ocupando mais 4 bytes. A chave em si começa após o cabeçalho.

Figura 10 – Identificando a posição da chave pública dentro do certificado gerado.

Fonte: De autoria própria.

Para preparar o certificado de acordo com o formato esperado, o *script* apresentado no Código 3.4 foi desenvolvido como parte do presente trabalho, retornando a *string* já pronta para ser inserida no código-fonte do Módulo de Defesa. O conteúdo do valor obtido deve ser colocado dentro da *array tcc_pub_key*, que guarda a chave pública a ser utilizada nas próximas etapas para realizar as

[2] <https://www.openssl.org/docs/man1.1.1/man1/openssl-asn1parse.html>

verificações. Se tudo foi feito corretamente, o conteúdo de *tcc_pub_key* deve começar com os bytes `0x3082010a`, como visto no exemplo demonstrativo apresentado pela Figura 11.

Código 3.4 – *Script* para extração da chave pública contida no certificado gerado.

```
#!/bin/sh

bitstring_line=$(openssl asn1parse -in "$1" -inform der | grep "BIT␣
   STRING" | head -1)
start=$(echo $bitstring_line | cut -d':' -f 1)
length=$(echo $bitstring_line | cut -d'␣' -f 2 | cut -d'=' -f 2)
sum=$(($start + $length + 2))
tail -c "+${sum}" $1 | xxd -p | tr -d '\n' | sed 's/\(..\)/\\x\1/g'
```

Figura 11 – Resultado da extração da chave pública usando o *script*.

```
(base) ➜  tcc git:(main) ✗ ./kernel-der-format.sh cert.der
\x30\x82\x01\x0a\x02\x82\x01\x01\x00\x94\xbe\x05\x53\x82\x41\x40\x32\x13\x0a\x93\x8b\x21\xd2\xb5\xa5\xcd\xcf\x7f\xe5\x31\x8e\
xa8\x25\x87\xec\x46\x7d\xd2\x6e\xa2\xb2\x16\x31\xe3\xbc\x52\xc8\x93\x42\x6e\x16\xc0\x90\xa4\xe3\x0a\x01\xbd\xb6\x66\xb1\x2e\x
2f\xcf\xe1\xd7\x6a\x9a\xfb\x40\xa7\x1d\x2e\x63\x69\x9e\x59\xaa\x19\x1d\xfa\xe9\x75\xcc\x95\x66\x32\xda\x02\x86\xf9\x4a\x54\x7
0\x47\xa1\x54\xd2\x3d\x9d\xed\x01\x67\x40\x13\x55\x15\x8c\xb1\x9a\x85\x9b\xeb\xcf\xfe\x8b\x1a\x8e\xe5\x1f\x61\x30\x09\x93\xc1
\x8a\x4d\x76\x02\x36\x26\xb5\x54\xc3\xe2\x6c\xee\x62\x45\x97\x3d\x0a\xc3\x22\x79\x50\xda\x6b\xec\xe7\x21\x64\xcf\x22\xc6\x90\
x14\x45\x35\x75\x4e\x85\x12\xdb\xcb\x14\x66\xe9\x34\x91\xdc\x98\xb9\xcc\x1c\x2c\x2b\x35\x28\xbd\x9a\xd3\x21\xce\xdf\xef\xd4\x
6f\x5c\x5b\xc4\xfa\x10\x61\x76\xc9\x9a\xe2\x7c\xbb\x16\xa5\x1a\x8e\x49\xe5\x46\x0f\x62\x69\x28\x1f\x03\xfb\xd4\x6b\x1c\xfa\x9
2\xde\x3f\x32\x85\x8d\x97\x64\x1e\x7c\xd3\x39\x3c\x04\xe4\x5c\xdb\x50\x62\x44\x31\x46\x5e\x13\x4b\xde\x99\x8b\xec\xc3\x9f\xae
\x0d\x7e\x7b\xd6\xc8\xaf\xa5\xf2\xd1\xd5\x46\xd3\x93\x10\x8b\x02\x03\x01\x00\x01\xa3\x53\x30\x51\x30\x1d\x06\x03\x55\x1d\x0e\
x04\x16\x04\x14\xe8\x81\x50\x4f\xbd\x65\x6c\x95\xcf\x81\x12\xb1\xdb\xad\x1f\xa5\x92\xc4\xac\x96\x30\x1f\x06\x03\x55\x1d\x23\x
04\x18\x30\x16\x80\x14\xe8\x81\x50\x4f\xbd\x65\x6c\x95\xcf\x81\x12\xb1\xdb\xad\x1f\xa5\x92\xc4\xac\x96\x30\x0f\x06\x03\x55\x1
d\x13\x01\x01\xff\x04\x05\x30\x03\x01\x01\xff\x30\x0d\x06\x09\x2a\x86\x48\x86\xf7\x0d\x01\x01\x0b\x05\x00\x03\x82\x01\x01\x00
\x1f\x4d\x9e\xc2\xb5\x43\x92\x28\xed\x03\x7b\x4d\xba\xb9\x59\xd2\x36\x97\x63\x5e\xb4\xf6\x30\xf5\xba\x35\xb4\x39\x60\x54\xf7\
x79\xff\xc0\xe7\x90\xb3\x45\xc4\xc6\xb9\x6b\x83\x13\x75\x6d\xee\xac\x8b\xe0\xc6\xe1\xe8\x3c\x23\x08\x05\x92\xb2\x98\xa5\x55\x
8c\xae\x9c\x5a\xa5\x0e\x12\x80\x98\x4e\xf1\x6f\x5d\x89\x38\xfc\x21\xea\x34\xcd\x72\xb4\x21\x18\xc1\x45\xe8\x6a\x9b\xe6\xe0\xb
5\x53\x07\xbe\xce\x43\x5f\xb5\x98\xdb\x80\xc4\x86\xc4\x79\x47\xeb\x0d\x99\xd9\x7d\x3c\x7e\x86\x95\xf3\x55\xdc\xa6\x20\x94\x3d
\x8e\x90\xee\xcd\x60\xf5\x86\xae\x89\x5f\x88\x07\x8b\x40\xc8\x07\xfd\x55\xb5\x3d\xc5\x0f\x13\xf2\x93\x2e\xe4\x16\xee\xa1\x23\
xde\x5a\xee\xe3\xbb\x58\x5b\xa8\x2c\xd8\xcd\xce\xd9\xe4\x7e\xfd\x26\x10\x57\x9c\x92\x47\x4e\x45\x63\xcb\xa1\x5d\x83\x79\x8b\x
2c\x35\x11\xe9\x51\x9d\x80\x36\x2a\x1b\x38\xa8\x22\xd0\xa4\x26\x4b\x5e\xe6\x23\x05\x86\xfc\x88\x89\x5e\x37\xda\x3d\x61\x9e\x7
a\xf1\xc8\xf7\x16\x69\xb8\x5d\xaa\x6b\xbb\xc6\xab\x61\x65\xc6\x4f\x2b\x02\x54\xc0\xc9\xbb\xe8\x7a\x1d\x7e\xb0\x63\xef\xa0\xa3
```

Fonte: De autoria própria.

3.2 Método de substituição na tabela de syscalls

O Módulo de Defesa foi projetado para ter sua atuação baseada na técnica de *syscall hijacking*, subvertendo esse método caracteristicamente presente em *rootkits* de kernel para uso com o objetivo de proteção. Sendo assim, o módulo irá fazer a alteração da tabela de chamadas de sistema, modificando as entradas relevantes para a instalação de métodos. Como discutido anteriormente no Capítulo 2.2, essas entradas são as *syscalls* `init_module` e `finit_module`. O primeiro passo é encontrar o endereço na memória em que a tabela (símbolo `sys_call_table`) se encontra.

Anteriormente à versão `5.7` o Linux exportava para os módulos o método `kallsyms_lookup_name()`, que podia ser usado para encontrar o endereço de um símbolo baseado apenas no seu nome. Por exemplo `kallsyms_lookup_name("sys_call_table")` podia ser utilizado para recuperar o endereço da tabela de *syscalls*. Contudo, nas versões mais recentes, o kernel parou de disponibilizar essa função por motivos de segurança. Já foram propostas outras formas para realizar esse processo mesmo no cenário em que a função não está disponível[3], porém, por motivos de simplicidade, a PoC do presente trabalho irá utilizar o 'arquivo' `/proc/kallsyms` para descobrir o endereço e

[3] <https://github.com/xcellerator/linux_kernel_hacking/issues/3>

adicioná-lo manualmente ao código-fonte. Isso é possível apenas caso o mecanismo de *Kernel Address Space Layout Randomization* (KASLR) esteja desabilitado, o que é verdadeiro para a maioria das distribuições Linux atualmente, visto que ele causa um elevado impacto na performance do kernel. O Código 3.5 mostra como a etapa de descoberta foi implementada no trabalho. A partir deste ponto o Módulo de Defesa está pronto para ser compilado.

Código 3.5 – Etapa de descoberta da posição da tabela de syscalls na memória do kernel.

```
static int __init tcc_init(void)
{
#if LINUX_VERSION_CODE < KERNEL_VERSION(5, 7, 0)
        sys_call_table = (sys_call_ptr_t *)kallsyms_lookup_name("
           sys_call_table");
#else
        sys_call_table = (sys_call_ptr_t *)0xffffffff826002c0;
#endif
    // ...
}
```

Em posse de uma referência para a tabela de *syscalls*, é realizada a substituição da entrada correspondente à chamada responsável pela adição de módulos ao kernel, salvando essa para que ela possa ser posteriormente invocada a partir do novo método que será injetado em seu lugar. Nesta Prova de Conceito, será feita apenas a substituição da chamada *finit_module*, porém *init_module* também deve ser modificada da mesma forma em uma implementação completa para garantir a proteção.

Com a função original salva, é possível então substituir a entrada correspondente. Para isso, primeiramente, é preciso modificar a página da memória onde a tabela de *syscalls* se encontra, pois essa é configurada com permissões de apenas leitura durante o *boot* do kernel, por motivos de segurança. Como o Módulo de Defesa executa em nível de supervisor, e portanto possui liberdade para realizar qualquer modificação ao sistema, podemos fazer o processo inverso e ativar permissões de escrita na região de memória por meio da função apresentada no Código 3.6.

Código 3.6 – Permitir escrita na página de memória onde a tabela está contida.

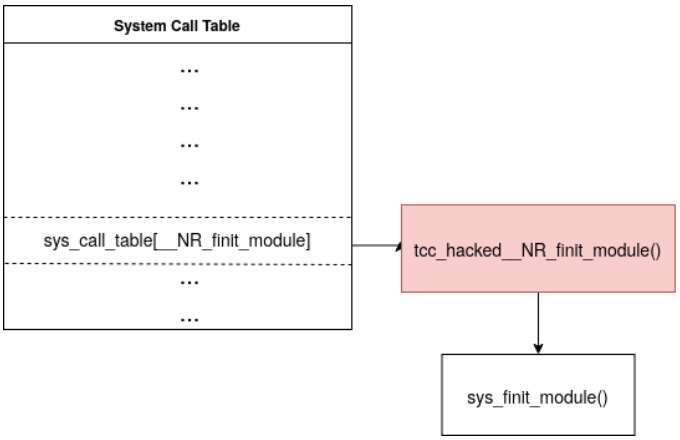
```
static void set_page_rw(unsigned long address)
{
        unsigned int level;
        pte_t *permissions = lookup_address(address, &level);
        pr_info("Setting␣Page␣Permissions␣to␣Read/Write\n");
        if (permissions->pte & ~_PAGE_RW)
                permissions->pte |= _PAGE_RW;
}
```

Outra opção possível nessa situação, em CPUs x86-64, é modificar o registrador *cr0* para habilitar a escrita em páginas que tenham apenas permissão de leitura, por meio da ativação da

flag `Write Protection` (WP). A escolha por uma ou outra é suficiente para fazer as modificações necessárias na tabela.

Com a permissão de escrita obtida, basta diretamente substituir a entrada por uma função customizada, `tcc_hacked__NR_finit_module`, de forma a finalizar a montagem do novo fluxo de execução. Essa técnica é conhecida como trampolim, onde um processamento adicional é injetado e realizado antes de continuar com a execução do fluxo original. No presente caso, a proposta é que o arquivo do módulo será verificado em relação à assinatura que deverá estar presente no seu fim, só então podendo prosseguir com a instalação. O diagrama da Figura 12 demonstra esse processo.

Figura 12 – Esquematização da técnica de trampolim para adicionar verificação de permissão antes da chamada original à *syscal*.



Fonte: De autoria própria.

3.3 Verificação da Assinatura

Para fazer a verificação da assinatura, necessita-se encontrá-la no binário e separar o conteúdo do arquivo do módulo original para fazer o cálculo da hash. Como discutido anteriormente, a assinatura se encontra no final do arquivo, consistindo de 256 bytes de dados (2048 bits pois a chave é do tipo RSA de 2048 bits). Apesar do formato ELF não conter informações sobre o tamanho de um arquivo, o kernel do Linux possui uma API para recuperá-lo a partir de um arquivo já aberto, representado pela `struct file`. Isso é feito a partir da função `i_size_read()`, a qual recebe um `inode`, que pode ser construído a partir da `struct file` utilizando a função `file_inode()`. A realização desses passos resultará no tamanho total do arquivo, ao qual precisamos subtrair o valor `256` (tamanho em bytes da assinatura) para encontrar o final do módulo em si, e a posição onde começa a assinatura. O processo descrito é realizado pelo Código 3.7.

Código 3.7 – Encontrando o final do módulo no binário.

```
asmlinkage long tcc_hacked__NR_finit_module(const struct pt_regs *
    regs)
{
        // ...

        f = fget(regs->di); // di = file descriptor
```

```
        // https://stackoverflow.com/a/70125102
        module_size_before_signature = i_size_read(file_inode(f)) -
           TCC_SIGNATURE_LENGTH;
        module_data = kmalloc(module_size_before_signature,
           GFP_KERNEL);
        kernel_read(f, module_data, module_size_before_signature, 0)
           ;

        // ...
}
```

A hash do módulo pode então ser calculada. A API criptográfica do Linux possui tudo que é necessário para realizar esta etapa, como pode ser visto no Código 3.8, que calcula o `SHA256` da região de memória passada como parâmetro. No Módulo de Defesa, essa região engloba o módulo sendo verificado, do *offset* 0 até antes do começo da assinatura.

Código 3.8 – Cálculo da hash do módulo sendo carregado.

```
static struct shash_desc *init_sdesc(struct crypto_shash *alg)
{
        struct shash_desc *sdesc;

        sdesc = kmalloc(sizeof(*sdesc) + crypto_shash_descsize(alg),
            GFP_KERNEL);
        if (!sdesc)
                return ERR_PTR(-ENOMEM);

        sdesc->tfm = alg;
        return sdesc;
}

static int tcc_calculate_hash(u8 *data, u64 length, u8 *digest)
{
        struct crypto_shash *alg;
        struct shash_desc *sdesc;
        char *hash_alg_name = "sha256";

        alg = crypto_alloc_shash(hash_alg_name, 0, CRYPTO_ALG_ASYNC)
           ;
        if (IS_ERR(alg)) {
                pr_err("can't␣alloc␣alg␣%s\n", hash_alg_name);
                return PTR_ERR(alg);
        }
```

```
        sdesc = init_sdesc(alg);
        if (IS_ERR(sdesc)) {
                pr_err("can't␣alloc␣sdesc\n");
                return PTR_ERR(sdesc);
        }

        crypto_shash_digest(sdesc, data, length, digest);

        kfree(sdesc);
        crypto_free_shash(alg);

        return 0;
}
```

Com a hash calculada, basta agora utilizar a API criptográfica do kernel, por meio da função `public_key_verify_signature()`, para verificar se a assinatura contida no módulo é idêntica à calculada. Esse método recebe duas estruturas, uma contendo informações sobre a chave pública, que foi adicionada estaticamente ao Módulo de Defesa anteriormente, e outra contendo informações sobre a hash calculada que espera-se ser validada. O Código 3.9 demonstra esse processo.

Código 3.9 – Verificação da assinatura em relação à hash calculada.

```
static int tcc_verify_signature(u8 *signature, size_t signature_size
    , u8 *expected_digest, size_t expected_digest_size)
{
        struct public_key rsa_pub_key = {
                .key = tcc_pub_key,
                .keylen = sizeof(tcc_pub_key),
                .pkey_algo = "rsa",
                .id_type = "X509"};

        struct public_key_signature sig = {
                .s = signature,
                .s_size = signature_size,
                .digest = expected_digest,
                .digest_size = expected_digest_size,
                .pkey_algo = "rsa",
                .hash_algo = "sha256",
                .encoding = "pkcs1"
        };

        int error = public_key_verify_signature(&rsa_pub_key, &sig);
        if (error) {
```

```
                pr_info("error verifying. error %d\n", error);
                return error;
        }

        pr_info("verified successfuly!!!\n");
        return 0;
}
```

Nesse ponto, caso a validação tenha sido bem sucedida, ou seja, caso a hash tenha sido verificada em relação a assinatura e confirmada sua equivalência, há a garantia de que o módulo é o mesmo que foi assinado originalmente a partir da chave privada a qual, pela arquitetura proposta, é conhecida apenas pelo real administrador do sistema, e portanto o módulo não foi alterado e a instalação pode continuar com segurança. Caso contrário, o Módulo de Defesa irá rejeitar a tentativa e retornar um erro para o espaço de usuário, mantendo o kernel seguro.

A solução proposta pressupõe que o administrador do sistema é também o que fará a instalação de módulos no futuro, sendo responsável pele processo de sua assinatura. Dessa forma, certificados ”auto-assinados” seriam suficientes visto que o usuário poderá garantir que ele mesmo é o emissor. Contudo, pode ser desejável garantir essa procedência de forma mais formal, por meio de um *Certificate Authority* (CA), ao qual pode realizar o processo de assinatura. Dessa forma, é possível verificar também a origem do módulo, tendo maior segurança que as chaves não foram alteradas em alguma etapa do processo, corrompendo-o. Este trabalho não aborda o tema mais a fundo além do explícito anteriormente, supondo que a assinatura própria será suficiente para o caso tratado.

3.3.1 Protegendo a chave privada

Para que a solução seja segura, é imprescindível que a chave privada seja mantida secreta e que não possa ser utilizada pelo usuário root. Dessa forma, manter a chave desprotegida no sistema é inseguro. Para garantir a segurança da chave, são utilizados alguns conceitos. O primeiro é o fato de que a chave privada não pode ser derivada a partir da chave pública (PREVEIL, 2021). Ou seja, mesmo que o atacante tenha acesso ao Módulo de Defesa, o qual contém o certificado, não será possível gerar a chave privada apenas com esse conhecimento.

É necessário também que a chave privada seja guardada de forma segura. Neste contexto, a primeira possibilidade que se visualiza é a de destruição da chave após o processo de assinatura. Isso garante que um hacker malicioso não conseguirá tomar controle do mecanismo de assinatura de módulos, porém também impede que o administrador o faça, tornando o Módulo de Defesa uma solução de uso único.

Outra opção é criptografar a chave privada por meio de uma senha. Mantém-se o arquivo no sistema sendo protegido, mas o fato de estar criptografada impede que o atacante possa utilizá-la, tornando o verdadeiro administrador, o qual possui a senha (de preferência guardada apenas em sua memória), o único capaz de assinar módulos.

Ainda no cenário em que a chave será mantida no mesmo sistema que está sendo protegido pelo Módulo de Defesa, há a possibilidade do uso de Trusted Platform Module (TPM), que é um elemento de hardware seguro disponível em alguns processadores em que chaves privadas são armazenadas e a realização de criptografias pode ser feita sem a necessidade de as expor (MICROSOFT, 2022b). É possível proteger esse sistema com o uso de PINs, o que torna essa solução segura para o contexto apresentado.

Por fim, há a solução de manter a chave em uma máquina externa, assinando os módulos nele antes de movê-los para o sistema sob proteção do Módulo de Defesa e instalá-los. Ainda há a necessidade de garantir a segurança da chave na outra máquina, mas a chave estará segura contra o ataque que comprometer o usuário root do sistema em análise.

3.4 Impedindo a remoção da proteção

Para impedir que a solução seja facilmente removida do kernel com o uso da *syscall* `delete_module`, essa também é substituída utilizando a mesma técnica de sequestro proposta para a inserção, de forma a executar uma verificação que impede que o Módulo de Defesa seja descarregado. No seu lugar, a função apresentada no Código 3.10 faz a verificação de se o módulo sendo removido é o Módulo de Defesa, por meio da comparação do seu nome. Em caso positivo, a chamada de sistema retorna um erro de permissão. Caso contrário, é permitida a continuação da remoção, não sendo necessária nenhuma verificação de assinatura. Considera-se que remover módulos não precisa ser validado pois não há a adição de código que pode ser malicioso.

Código 3.10 – Função a ser executada antes da remoção de módulos.

```
asmlinkage long tcc__NR_delete_module(const struct pt_regs *regs)
{
        char name[MODULE_NAME_LEN];
        char __user *name_user = regs->di;

        if (strncpy_from_user(name, name_user, MODULE_NAME_LEN - 1)
           < 0) {
                return -EFAULT;
        }

        if (strncmp(name, "tcc", strlen(name)) == 0) {
                pr_warn("Can't␣remove␣tcc␣module␣;)\n");
                return -EPERM;
        }

        return original_delete_module(regs);
}
```

4 Metodologia

4.1 Ambiente de Testes

Para este trabalho, utilizou-se o kernel do Linux versão 5.4.190[1], compilado para a arquitetura x86-64 com o uso do compilador GCC 12.1.0 e GNU Make 4.3. O arquivo de configuração utilizado encontra-se disponível no Github do autor[2].

Foi decidido pela utilização de um ambiente de testes emulado, com o uso da ferramenta de virtualização QEMU versão 6.2.0. Para isso, foi necessária a criação de uma imagem do sistema *guest* por meio do *script*[3] de criação de imagens disponível no repositório do projeto Google Syzkaller. Esse *script* gera uma imagem baseada na distribuição Ubuntu, incluindo a adição da chave para realizar SSH para dentro da máquina virtual, permitindo uma configuração rápida do ambiente.

A execução do ambiente foi realizada por meio do comando apresentado no Código 4.1, Sendo `image` a pasta em que o *script* `create-image.sh` foi executado, gerando o arquivo `stretch.img` visto no comando.

Código 4.1 – Script de execução do ambiente de testes

```
qemu-system-x86_64 \
        -m 4G \
        -smp 1 \
        -kernel arch/x86/boot/bzImage \
        -append "console=ttyS0 root=/dev/sda nokaslr" \
        -drive file=image/stretch.img,format=raw \
        -net user,host=10.0.2.10,hostfwd=tcp:127.0.0.1:10021-:22 \
        -net nic,model=e1000 \
        -enable-kvm \
        -nographic \
        -pidfile vm.pid \
        2>&1 | tee vm.log
```

Para a geração das chaves criptográficas e certificados, fez-se o uso do OpenSSL 1.1.0l, presente na imagem criada com o *script*.

4.2 Experimentos

Para testar a eficácia da solução, três testes foram realizados, além de um quarto para verificar a garantia da integridade da mesma. Em todos os casos, um simples módulo *Hello World* foi utilizado,

[1] <https://mirrors.edge.kernel.org/pub/linux/kernel/v5.x/linux-5.4.190.tar.xz>
[2] <https://gist.github.com/vccolombo/8d325955f43df699c4b430acbb86ae07>
[3] <https://github.com/google/syzkaller/blob/0fc5c330fea4b4129567aaa44ea5a134cb850bbb/tools/create-image.sh>

cujo conteúdo pode ser visualizado no Código 4.2.

Código 4.2 – Módulo de teste

```
#include <linux/init.h>
#include <linux/module.h>

MODULE_AUTHOR("Victor␣Colombo");
MODULE_LICENSE("Dual␣BSD/GPL");

static int hello_init(void)
{
        printk(KERN_ALERT "Hello,␣world\n");
        return 0;
}

static void hello_exit(void)
{
        printk(KERN_ALERT "Obrigado␣por␣ler␣meu␣TCC␣;)\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

4.2.1 Caso 1.1: LKM sem qualquer assinatura

O primeiro teste consiste na tentativa de instalação de um módulo que não possui assinatura no local esperado. Isso significa que o módulo pode até conter uma assinatura proveniente dos repositórios oficiais, mas é necessário que haja também uma assinatura proveniente do método proposto nesse trabalho. O resultado esperado nesse caso é que o módulo deve ser rejeitado, não sendo instalado.

4.2.2 Caso 1.2: LKM com assinatura proveniente de uma chave privada incorreta

Em seguida, o teste com um módulo assinado com o método proposto nesse trabalho, porém com a chave privada incorreta (não equivalente ao certificado presente no Módulo de Defesa). Aqui, também, espera-se que sua instalação seja rejeitada, visto que as assinaturas não batem.

4.2.3 Caso 1.3: LKM com assinatura proveniente da chave privada original

O último caso consiste da instalação bem sucedida: o módulo de teste é assinado com a chave privada correta, relacionada ao certificado presente no Módulo de Defesa. Dessa forma, a verificação deve checar que o módulo a ser instalado possui a assinatura correta e portanto é autorizado a ser instalado.

4.2.4 Caso 2.1: Bloqueio da remoção do Módulo de Defesa

Para garantir que não seja possível remover a proteção com facilidade, a remoção do Módulo de Defesa deve ser desabilitada, retornando um erro e mantendo o mesmo carregado.

5 Resultados

Nesta seção será avaliado o comportamento do Módulo de Defesa em relação aos testes propostos no tópico de Metodologia. Quando instalada, a defesa deverá proibir a instalação de módulos nos cenários em que não houve assinatura ou que essa foi gerada a partir de uma chave incorreta, não correspondente à chave pública compilada estaticamente como parte da proteção. O único caso em que a instalação deve prosseguir é quando o módulo foi corretamento assinado com a chave privada correspondente ao certificado presente no Módulo de Defesa. Além disso, para garantir a integridade da defesa, qualquer tentativa de remoção do Módulo de Defesa deve ser rejeitada, enquanto outros módulos podem ser removidos sem dificuldades.

5.1 Caso 1.1: LKM sem qualquer assinatura

No primeiro caso, era esperado que um LKM sem qualquer assinatura fosse rejeitado pela proteção desenvolvida. Como pode ser visto na Figura 13, o resultado esperado foi obtido, tendo a instalação sido rejeitada, e o módulo consequentemente não aparece na lista de carregados no kernel.

Figura 13 – Módulo de Defesa recusando a instalação de módulos não assinados.

```
~# insmod hello.ko
insmod: ERROR: could not insert module hello.ko: Operation not permitted
~# [  222.520787] error verifying. error -22
lsmod
Module                  Size  Used by
tcc                    16384  0
```

Fonte: De autoria própria.

5.2 Caso 1.2: LKM com assinatura proveniente de uma chave privada incorreta.

No segundo caso, era esperado que um LKM assinado com uma chave privada não correspondente à chave pública do certificado fosse rejeitado pela proteção desenvolvida. Como pode ser visto na Figura 14, o resultado esperado foi obtido, tendo a instalação sido rejeitada, e o módulo também não aparece na lista de carregados no kernel.

Figura 14 – Módulo de Defesa recusando a instalação de módulos assinados com a chave privada não correspondente à chave pública

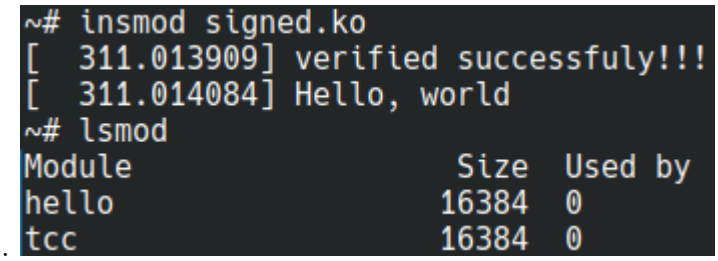
```
~# insmod bad-signed.ko
insmod: ERROR: could not insert module bad-signed.ko: Operation not permitted[   11.445835] error verifying. error -22

~# lsmod
Module                  Size  Used by
tcc                    16384  0
```

Fonte: De autoria própria.

5.3 Caso 1.3: LKM com assinatura proveniente da chave privada original.

O terceiro caso contém o módulo preparado corretamente para ser utilizado com a proteção desenvolvida. O LKM assinado com a chave privada original, nesse contexto pertencente ao verdadeiro administrado de um sistema a ser defendido, é instalado no Linux que contém o Módulo de Defesa carregado. O resultado é apresentado na Figura 15, onde pode ser visto que o carregamento ocorreu com sucesso e o módulo é mostrado pelo comando *lsmod*.

Figura 15 – Módulo de Defesa autorizando a instalação de módulos corretamente assinados

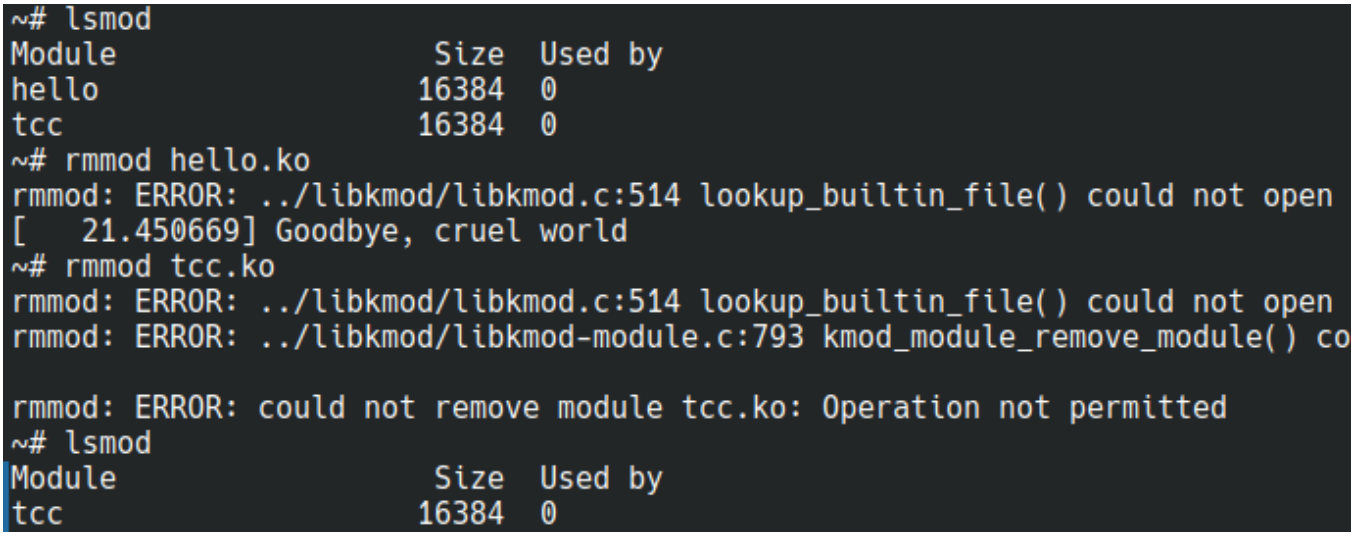


Fonte: De autoria própria.

5.4 Caso 2.1: Bloqueio da remoção do Módulo de Defesa.

Finalmente, para verificar que a proteção continuará imaculado ao tentar removê-lo, o último teste explora o cenário em que o comando *rmmod* é executado com o Módulo de Defesa como alvo, possivelmente por um agente malicioso buscando afetar o sistema. A Figura 16 mostra o seu resultado, demonstrando que não foi possível remover o módulo por esse método.

Figura 16 – Módulo de Defesa recusando a própria remoção.



Fonte: De autoria própria.

5.5 Discussão

Como pode ser visto acima, o Módulo de Defesa implementado nesse trabalho foi bem sucedido em proteger contra as situações hipotéticas propostas. No cenário em que a funcionalidade

de instalação de módulos no kernel Linux está sendo explorada como vetor de ataque para alcançar a persistência no sistema, o mecanismo apresentado se mostra uma defesa válida ao impedir que um agente malicioso que obteve acesso ao usuário privilegiado por meio de uma escalada de privilégios possa comprometer o sistema por meio da instalação de um *rootkit*.

Como citado no desenvolvimento do trabalho, a Prova de Conceito apresentada utiliza diversos atalhos na sua implementação que não tornam a defesa ideal para utilização em cenários reais. Por exemplo, não foram estudadas maneiras de integrar a proteção a algum repositório de pacotes das distribuições, sendo que a instalação de programas que sejam compostos de módulos nesses casos falhariam, visto que o Módulo de Defesa bloquearia seu carregamento no kernel. Ainda, o fato de haver conflito nas proteções caso o módulo seja assinado tanto com a ferramenta do próprio Linux, quanto com a proposta aqui, de forma que (pelo menos teoricamente) apenas a que foi adicionada por último seria utilizável. Por fim, identifica-se também a questão do bloqueio de remoção da proteção ser uma checagem estática de seu nome, o que é uma premissa frágil, mas cujo impacto e possível contorno da defesa não foi analisado.

Não obstante, é importante ressaltar que a proteção proposta, por si só, peca ao não lidar com todas as vias de ataque que podem ser utilizadas pelo usuário *root* para realizar alterações no kernel. Por exemplo, é possível o uso do mecanismo de `ftrace` para modificar o código do kernel sem a necessidade de reiniciar o sistema, a partir do mecanismo de *Livepatch* (FREE SOFTWARE FOUNDATION, 2019). Ainda, há o `eBPF`, que além de sua capacidade de *tracing* também pode ser utilizado para realizar a filtragem de *syscalls* que chegam ao kernel, já tendo sido matéria de análise em relação ao seu uso para a proteção contra *rootkits*, como pode ser visto em (MAOUDA, 2022).

Inclusive, os mecanismos citados anteriormente poderiam ter sido escolhidos como base da implementação da Prova de Conceito desse trabalho. A escolha pela escrita de um módulo, por meio da intricada técnica de sequestro de chamadas de sistema, se deu pelo interesse em propor uma proteção ao cenário em que se deseja manter essa funcionalidade ativa, ao mesmo tempo que não se quer causar interrupção no sistema para utilizar o mecanismo já implementado de assinaturas no Linux. Dessa forma, pode-se supor que existam outros mecanismos de defesa protegendo outras interfaces críticas com o kernel, ou que essas interfaces estejam desabilitadas, e portanto não seriam um risco para o Módulo de Defesa. Ainda, o uso de um kernel compilado estaticamente pode servir como técnica de mitigação, mas ao mesmo tempo limita a extensão de suas funcionalidades com o uso dos módulos, fazendo-se necessário uma nova compilação caso haja esse interesse.

Contudo, além das interfaces discutidas anteriormente, outras questões também precisam ser levadas em consideração. A primeira é o fato de o Módulo de Defesa ser frágil em um cenário em que o sistema possa ser reiniciado pelo atacante, o que acarretaria na remoção da proteção. Mesmo se o administrador configurá-lo de forma a ser carregado durante o *boot*, o Linux não garante a ordem de carregamento de módulos durante a inicialização do sistema, de forma que um atacante pode simplesmente criar um módulo que por acaso virá a ser carregado antes da proteção. Uma pesquisa aprofundada de como contornar essa limitação precisaria ser realizada para permitir a utilização eficiente da solução aqui proposta.

Outra vulnerabilidade que não é exclusiva da solução apresentada, mas pode colocá-la em

risco, é a modificação do binário do kernel em si. Um hacker malicioso pode substituí-lo e afetar o sistema em uma reinicialização posterior mesmo no caso em que o Módulo de Defesa seja carregado na inicialização do sistema. Esse cenário normalmente é mitigado pelo uso do Secure Boot ou SELinux, mas caso estes não estejam disponíveis, a mesma técnica de sequestro de chamadas de sistema apresentada neste trabalho poderia ser utilizada para proteger também arquivos sensíveis (como a imagem do kernel, por exemplo) contra tentativas de escrita não autorizada.

A solução proposta pode ser utilizada no contexto em que a recompilação para adição da funcionalidade já existente não seja possível. Um exemplo pertinente deste cenário é o de distribuições Linux, tais como Ubuntu, em que o kernel é modificado e configurado de forma ideal para funcionar com os programas e pacotes existentes no sistema. Nesta situação, a imagem do sistema já é enviado para os usuários, instalado e atualizado de forma praticamente automática. Modificar e compilar o próprio kernel causa então uma ruptura neste modelo, de forma que o sistema não mais receberá atualizações automáticas da versão do núcleo, tornando responsabilidade do usuário realizar esse processo de forma manual para garantir que novas funcionalidades e correções sejam aplicadas ao kernel customizado. O Módulo de Defesa evita esse problema, já que não modifica o binário do sistema em si, permitindo que continue a receber as atualizações providas pela distribuição.

Outro contexto de interesse é o de aparelhos que não são mais suportados pelos seus fabricantes, como por exemplo dispositivos embarcados antigos. Neste cenário, a atualização do kernel costuma não ser viável devido às modificações que foram feitas pelo fabricante. O uso do Módulo de Defesa então pode ser viável para acrescentar uma camada de defesa nesses dispositivos.

6 Conclusão

Como visto ao decorrer do trabalho, o Linux se apresenta como um sistema operacional de grande importância na atualidade, no qual a maior parte da infraestrutura crítica dos governos e grandes corporações é baseada. Dessa forma, é primordial a atenção quanto à sua proteção contra a exploração de vulnerabilidades por agentes maliciosos, sendo a utilização do mecanismo de módulos para esconder a presença de *malwares* no sistema uma questão preocupante.

Dessa forma, o presente trabalho propôs a implementação de um método para impedir a instalação de módulos a não ser pelo verdadeiro administrador do sistema, utilizando-se do conceito de assinatura de binários. Semelhante ao mecanismo já existente no Linux atualmente, a defesa projetada diferencia-se pelo fato de procurar permitir a adição da proteção mesmo em um kernel compilado sem as configurações pertinentes, evitando a interrupção do serviço e a complexidade de configuração que normalmente está acompanhada desse processo. Isso se deu pelo uso do próprio mecanismo de adição de módulos ao Linux, que foi utilizado como base para inserir as modificações no código do kernel, modificando o funcionamento das chamadas de sistema pertinentes à instalação de outros módulos, e portanto de *rootkits*, acrescentando-se a verificação de uma assinatura que, pela arquitetura proposta, pode ser proveniente apenas do dono do sistema.

Em relação aos objetivos específicos propostos, discorreu-se sobre como a criptografia assimétrica pode ser utilizada para verificação da origem de mensagens no geral, e mais especificamente de binários de software, por meio do conceito de assinaturas, incluindo de forma mais aprofundada a assinatura de módulos no Linux. Como principal objeto de trabalho e contribuição desse projeto, foi proposto e implementado um Módulo de Defesa, que pode ser instalado nos sistemas com a finalidade de protegê-lo contra a instalação indevida de LKMs por agentes maliciosos, como discutido no parágrafo anterior. Finalmente, todo o ferramental elaborado e a explicação sobre o uso da API criptográfica do kernel Linux, que carece de documentação e exemplos concretos nos manuais e na internet em geral, está presente como contribuição para aqueles a qual possa interessar.

A Prova de Conceito implementada mostrou-se eficiente no bloqueio das ameaças simuladas propostas. Três casos de teste foram propostos, incluindo o cenário de ausência de assinatura no módulo sendo instalado, presença de assinatura incorreta, e presença de assinatura válida e compatível. A proteção obteve êxito nos três cenários, impedindo a instalação nos dois primeiros, e permitindo no terceiro. Como verificação à integridade do Módulo de Defesa, mais um teste foi executado, dessa vez buscando remover o mecanismo do kernel com o uso de `rmmod`, o que também foi bloqueado com êxito, indicando que, ressalvados os problemas abordados acima, houve proteção satisfatória ao sistema.

Como caminho para trabalhos futuros, propõe-se a exploração mais aprofundada de temas que foram tangenciados na implementação atual, como a integração com repositórios de pacotes das distribuições, e a análise de riscos à defesa implementada como o uso de `ftrace` e `eBPF`, além de testes utilizando exemplos reais de *rootkits*.

Referências

0XAX. *System calls in the Linux kernel. Part 2.* <https://0xax.gitbooks.io/linux-insides/content/SysCall/linux-syscall-2.html>: [s.n.], 2020. [Online. Acesso em 20 de Julho, 2022]. Citado na página 21.

BOVET, D. P.; CESATI, M. *Understanding the Linux Kernel: from I/O ports to process management.* [S.l.]: "O'Reilly Media, Inc.", 2005. Citado na página 19.

CORBET, J.; RUBINI, A.; KROAH-HARTMAN, G. *Linux device drivers.* [S.l.]: "O'Reilly Media, Inc.", 2005. Citado 2 vezes nas páginas 15 e 24.

DEBIAN. *Debian Wiki SecureBoot.* <https://wiki.debian.org/SecureBoot>, 2022. [Online. Acesso em 30 de Julho, 2022]. Citado na página 32.

FREE SOFTWARE FOUNDATION. *Kernel module signing facility.* Versão 4.15. <https://www.kernel.org/doc/html/v4.15/admin-guide/module-signing.html>, 2018. [Online. Acesso em 20 de Julho, 2022]. Citado 2 vezes nas páginas 31 e 32.

FREE SOFTWARE FOUNDATION. *Livepatch.* Versão 5.3. <https://www.kernel.org/doc/html/v5.3/livepatch/livepatch.html>, 2019. [Online. Acesso em 08 de Agosto, 2022]. Citado na página 51.

FREE SOFTWARE FOUNDATION. *Linux Programmer's Manual elf(5).* Versão 5.13. <https://man7.org/linux/man-pages/man5/elf.5.html>, 2021. [Online. Acesso em 19 de Julho, 2022]. Citado na página 28.

FREE SOFTWARE FOUNDATION. *Linux Programmer's Manual init_module(2).* Versão 5.13. <https://man7.org/linux/man-pages/man2/init_module.2.html>, 2021. [Online. Acesso em 20 de Julho, 2022]. Citado na página 25.

FREE SOFTWARE FOUNDATION. *Linux Programmer's Manual syscalls(2).* Versão 5.13. <https://man7.org/linux/man-pages/man2/syscalls.2.html>, 2021. [Online. Acesso em 20 de Julho, 2022]. Citado na página 22.

FREE SOFTWARE FOUNDATION. *Linux Security Modules: General Security Hooks for Linux.* Versão 5.19. <https://www.kernel.org/doc/html/v5.19/security/lsm.html>, 2022. [Online. Acesso em 11 de Agosto, 2022]. Citado na página 26.

HOGLUND, G.; BUTLER, J. *Rootkits: subverting the Windows kernel.* [S.l.]: Addison-Wesley Professional, 2006. Citado na página 27.

IBM. *Digital signature overview.* <https://www.ibm.com/docs/en/b2badv-communication/1.0.0?topic=overview-digital-signature>, 2021. [Online. Acesso em 28 de Julho, 2022]. Citado na página 30.

KUMAR, G. *Linux Kernel Module : Internals of insmod and rmmod.* <https://gomathikumar1006.blogspot.com/2013/09/linux-kernel-module-internals-of-insmod.html>: [s.n.], 2013. [Online. Acesso em 13 de Julho, 2022]. Citado na página 25.

LOVE, R. *Linux kernel development.* [S.l.]: Pearson Education, 2010. Citado 3 vezes nas páginas 19, 20 e 21.

MAOUDA, I. *Hunting Rootkits with eBPF: Detecting Linux Syscall Hooking Using Tracee.* <https://blog.aquasec.com/linux-syscall-hooking-using-tracee>, 2022. [Online. Acesso em 08 de Agosto, 2022]. Citado na página 51.

MDN WEB DOCS. *Object-oriented programming*. <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming>, 2022. [Online. Acesso em 23 de Julho, 2022]. Citado na página 24.

MICROSOFT. *Microsoft Windows Hardware Developer docs Secure Boot*. <https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-secure-boot>, 2022. [Online. Acesso em 30 de Julho, 2022]. Citado na página 32.

MICROSOFT. *Trusted Platform Module Technology Overview*. <https://learn.microsoft.com/en-us/windows/security/information-protection/tpm/trusted-platform-module-overview>, 2022. [Online. Acesso em 18 de Setembro, 2022]. Citado na página 43.

PAUL, E. *What is Digital Signature: How it works, Benefits, Objectives, Concept*. <https://www.emptrust.com/blog/benefits-of-using-digital-signatures/>, 2017. [Online. Acesso em 28 de Julho, 2022]. Citado na página 30.

PREVEIL. *Public – private key pairs & how they work*. <https://www.preveil.com/blog/public-and-private-key/>, 2021. [Online. Acesso em 18 de Setembro, 2022]. Citado na página 42.

SALZMAN, P. J.; BURIAN, M.; POMERANTZ, O. *The linux kernel module programming guide*. 2007. Citado na página 25.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Applied operating system concepts*. [S.l.]: John Wiley & Sons, Inc., 1999. Citado 2 vezes nas páginas 20 e 24.

SPARKS, S.; BUTLER, J. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan*, v. 11, n. 63, p. 504–533, 2005. Citado na página 27.

TANENBAUM, A. *Modern operating systems*. [S.l.]: Pearson Education, Inc.„ 2009. Citado 2 vezes nas páginas 15 e 19.

TIS COMMITTEE ET AL. *Tool interface standard (TIS) executable and linking format (ELF) specification version 1.2*. [S.l.]: May, 1995. <https://refspecs.linuxbase.org/elf/elf.pdf>. [Online. Acesso em 02 de Julho, 2022]. Citado na página 28.

WELSH, M.; DALHEIMER, M. K.; KAUFMAN, L. *Running Linux*. [S.l.]: O'Reilly & Associates, Inc., 1999. Citado na página 19.

WILKINS, R.; RICHARDSON, B. Uefi secure boot in modern computer security solutions. In: *UEFI forum*. [S.l.: s.n.], 2013. p. 1–10. [Online. Acesso em 03 de Julho, 2022]. Citado na página 32.

APÊNDICE A – Código-fonte do Módulo de Defesa

Código A.1 – Código fonte completo do Módulo de Defesa.

```
#include <crypto/akcipher.h>
#include <crypto/hash.h>
#include <crypto/public_key.h>
#include <linux/stat.h>
#include <linux/syscalls.h>
#include <linux/version.h>

MODULE_AUTHOR("Victor␣Colombo");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("TCC␣Kernel␣Module␣-␣Protect␣against␣malicious␣
   modules␣installation");

#define MAX_SECTION_NAME_LENGTH 0x64
#define TCC_SIGNATURE_LENGTH 256
#define TCC_HASH_ALG_DIGEST_SIZE 32

unsigned char tcc_pub_key[] =
    "\x30\x82\x01\x0a\x02\x82\x01\x01\x00\xbf\x6f\x98\x0f\x35\x88\
       xf0\x9d\x04\xc9\x7c\x0f\xa8"
    "\xb6\x99\x4a\x3c\x45\x1d\x06\x64\xa6\x83\xcb\x4c\x68\x46\xc7\
       xd5\xb8\x8f\xd3\xfa\xdc\x79\x0c"
    "\x8e\xa1\x9a\x9f\xc6\x09\x91\x2c\xa5\x23\xc9\x48\x1e\xbb\x26\
       x76\xbb\xe3\x72\x61\x22\x3f\x24"
    "\xd1\x54\xbf\x13\x6d\x3f\x49\x9a\x44\x99\x68\x59\x10\x2d\x7c\
       xc7\x33\x44\xb7\x0a\x62\xcc\x31"
    "\xa9\x2a\x5b\x08\x67\xdd\xd8\x45\xc6\x64\x25\x46\x91\x01\x1a\
       x10\xec\xca\x5f\x17\x50\x36\x00"
    "\x85\xbd\x0c\x1d\x66\x0c\x58\x31\xbd\xc6\xc9\xd1\xf1\x8c\xf2\
       xad\xd7\x15\x99\xbe\x32\x6e\xc2"
    "\xf0\xee\xc6\x69\x7b\xb6\x0a\x27\xb9\xb9\x3c\x14\x39\xd4\xe2\
       x50\x05\xf8\x85\xe6\xd9\xac\xc6"
    "\x92\xda\x9a\x92\x39\xd0\x4d\x1c\x14\x92\x0e\xa3\x65\x3c\x39\
       x69\x4d\xe5\xa2\x06\xc4\xa0\x7d"
    "\x41\xdf\xeb\x8d\x21\xf2\x32\xfa\x58\x2e\xf1\xa6\x0b\x7e\x79\
```

```
    x95\xfc\x6e\x8d\xf1\x35\xf6\xb0"
  "\x8f\x29\x6d\x39\x41\x78\xb2\x40\xbc\x0a\xe1\x32\xb1\xff\x79\
    xc2\x84\x21\x08\x3c\xa6\xad\x75"
  "\x66\x16\x4c\x5a\x70\xc0\x4d\x57\xba\xa3\x02\xd3\x68\xd3\xc9\
    x8b\x87\x9b\xb2\x3b\xd7\xbf\x09"
  "\xdf\x2f\x5c\x3f\xd0\x29\x6f\xc8\xc2\xd6\x53\xf7\xa3\x02\x03\
    x01\x00\x01\xa3\x53\x30\x51\x30"
  "\x1d\x06\x03\x55\x1d\x0e\x04\x16\x04\x14\xbb\x48\xee\xe9\x01\
    xe8\x40\x85\x1e\x3e\xf4\x74\xcd"
  "\x29\x4a\xd6\xb0\x9e\xab\xb5\x30\x1f\x06\x03\x55\x1d\x23\x04\
    x18\x30\x16\x80\x14\xbb\x48\xee"
  "\xe9\x01\xe8\x40\x85\x1e\x3e\xf4\x74\xcd\x29\x4a\xd6\xb0\x9e\
    xab\xb5\x30\x0f\x06\x03\x55\x1d"
  "\x13\x01\x01\xff\x04\x05\x30\x03\x01\x01\xff\x30\x0d\x06\x09\
    x2a\x86\x48\x86\xf7\x0d\x01\x01"
  "\x0b\x05\x00\x03\x82\x01\x01\x00\x8e\xba\x06\x4a\xb4\xe5\x1d\
    x95\x35\xef\x8b\x89\xa0\x8c\x25"
  "\x18\x44\xbc\x2c\xe7\x01\x3d\x4f\xfb\x0f\x52\x4a\x96\x60\x93\
    x06\x78\xd5\x0f\x69\xdc\xa0\xf4"
  "\x4b\xdc\x48\x1e\x73\x4f\x3d\x75\xc7\xfc\xa9\xc4\xbc\x2e\xfb\
    x02\x41\x5b\x9d\x18\x89\xc9\x05"
  "\xaf\x44\xaf\xef\x3f\xb3\x5c\x8f\xc5\x9f\x30\xd0\x19\xa1\xf8\
    x28\x15\xc3\xb8\xfd\x03\x02\xab"
  "\xc8\xb2\x1e\x09\xdc\xe7\x88\x09\x41\xfb\x1b\xc1\x0e\x07\x0d\
    xea\x87\x75\xd5\x71\x33\xfc\x00"
  "\x9d\x24\xb0\x5c\x4d\x92\xf3\x4a\x7d\x32\xde\x3b\xc3\x72\x88\
    x80\xd1\xc6\x46\x2b\x7f\xd5\xa0"
  "\xba\xc7\x53\x0f\x26\x77\x3f\x74\xf6\x09\x7e\x86\x74\x30\xe2\
    xcd\xe9\xf7\x6b\x13\x4d\xa4\x6f"
  "\x82\x73\xd3\x83\xae\x1b\x1f\x99\x1a\x06\x09\x81\x26\x19\x1e\
    x41\xb9\x21\x52\x94\x5d\xcd\xf3"
  "\xa6\x0f\x60\x8e\x11\xa0\xaf\xff\x1e\x3d\xbc\xd8\xe4\xbb\xa9\
    xdf\xd1\x23\xf6\x5f\x2a\xea\x95"
  "\xcc\xf1\x49\xe8\x3e\x5e\xc0\xcb\xd3\x1b\x8b\x57\x03\x6e\x8d\
    xf2\xe3\x45\x8a\x20\x1c\x01\xe1"
  "\x9d\x2a\x4c\x2a\xab\x93\xde\x3e\x11\xed\xc7\xf0\xb2\x77\x77\
    x40\x10\x96\x02\xba\x51\xa2\x57"
  "\x5b\x11\x23\x33\xca\x8a\x5e\xde\xba\x6c\x01";

static unsigned long tcc_orig_cr0;

inline void tcc_write_cr0(unsigned long cr0) { asm volatile("mov␣
```

```
    %0,%%cr0" : "+r"(cr0)); }

// https://infosecwriteups.com/linux-kernel-module-rootkit-syscall-
    table-hijacking-8f1bc0bd099c
static inline void tcc_unprotect_memory(void)
{
        tcc_orig_cr0 = read_cr0();
        tcc_write_cr0(tcc_orig_cr0 & (~0x10000)); /* Set WP flag to
           0 */
}

static inline void tcc_protect_memory(void) { tcc_write_cr0(
    tcc_orig_cr0); /* Set WP flag to 1 */ };

#define NO_PROTECTION(X)        \
        do {                            \
                preempt_disable();      \
                tcc_unprotect_memory(); \
                X;                      \
                tcc_protect_memory();   \
                preempt_enable();       \
        } while (0)

static void set_page_rw(unsigned long address)
{
        unsigned int level;
        pte_t *permissions = lookup_address(address, &level);
        pr_info("Setting␣Page␣Permissions␣to␣Read/Write\n");
        if (permissions->pte & ~_PAGE_RW)
                permissions->pte |= _PAGE_RW;
}

static void set_page_ro(unsigned long address)
{
        unsigned int level;
        pte_t *permissions = lookup_address(address, &level);
        pr_info("Setting␣Page␣Permissions␣to␣Read-Only\n");
        permissions->pte &= ~_PAGE_RW;
}

// https://stackoverflow.com/questions/63742987/hashing-same-string-
    generates-different-results-in-kernel-module
```

```
static struct shash_desc *init_sdesc(struct crypto_shash *alg)
{
        struct shash_desc *sdesc;

        sdesc = kmalloc(sizeof(*sdesc) + crypto_shash_descsize(alg),
            GFP_KERNEL);
        if (!sdesc)
                return ERR_PTR(-ENOMEM);

        sdesc->tfm = alg;
        return sdesc;
}

static int tcc_calculate_hash(u8 *data, u64 length, u8 *digest)
{
        struct crypto_shash *alg;
        struct shash_desc *sdesc;
        char *hash_alg_name = "sha256";

        alg = crypto_alloc_shash(hash_alg_name, 0, CRYPTO_ALG_ASYNC)
           ;
        if (IS_ERR(alg)) {
                pr_err("can't␣alloc␣alg␣%s\n", hash_alg_name);
                return PTR_ERR(alg);
        }

        sdesc = init_sdesc(alg);
        if (IS_ERR(sdesc)) {
                pr_err("can't␣alloc␣sdesc\n");
                return PTR_ERR(sdesc);
        }

        crypto_shash_digest(sdesc, data, length, digest);

        kfree(sdesc);
        crypto_free_shash(alg);

        return 0;
}

static int tcc_verify_signature(u8 *signature, size_t signature_size
   , u8 *expected_digest,
```

```
                                size_t expected_digest_size)
{
        struct public_key rsa_pub_key = {.key = tcc_pub_key,
                                         .keylen = sizeof(
                                           tcc_pub_key),
                                         .pkey_algo = "rsa",
                                         .id_type = "X509"};

        struct public_key_signature sig = {.s = signature,
                                           .s_size = signature_size,
                                           .digest = expected_digest
                                             ,
                                           .digest_size =
                                             expected_digest_size,
                                           .pkey_algo = "rsa",
                                           .hash_algo = "sha256",
                                           .encoding = "pkcs1"};

        int error = public_key_verify_signature(&rsa_pub_key, &sig);
        if (error) {
                pr_info("error␣verifying.␣error␣%d\n", error);
                return error;
        }

        pr_info("verified␣successfuly!!!\n");
        return 0;
}

typedef asmlinkage long (*sys_call_ptr_t)(const struct pt_regs *);
static sys_call_ptr_t *sys_call_table;
sys_call_ptr_t original_finit_module;

asmlinkage long tcc_hacked__NR_finit_module(const struct pt_regs *
   regs)
{
        int err, header_size;
        struct file *f;
        u64 module_size_before_signature;
        u8 *calculated_hash, *tcc_signature_section_data, *
           module_data;

        // struct file, defined in <linux/fs.h>
```

```
        f = fget(regs->di); // di = file descriptor

        // https://stackoverflow.com/a/70125102
        module_size_before_signature = i_size_read(file_inode(f)) -
           TCC_SIGNATURE_LENGTH;
        module_data = kmalloc(module_size_before_signature,
           GFP_KERNEL);
        kernel_read(f, module_data, module_size_before_signature, 0)
           ;

        calculated_hash = kmalloc(TCC_HASH_ALG_DIGEST_SIZE,
           GFP_KERNEL);
        err = tcc_calculate_hash(module_data,
           module_size_before_signature, calculated_hash);
        if (err) {
                goto out;
        }

        f->f_pos = module_size_before_signature;
        tcc_signature_section_data = kmalloc(TCC_SIGNATURE_LENGTH,
           GFP_KERNEL);
        kernel_read(f, tcc_signature_section_data,
           TCC_SIGNATURE_LENGTH, &f->f_pos);

        err = tcc_verify_signature(tcc_signature_section_data,
           TCC_SIGNATURE_LENGTH,
                                   calculated_hash,
                                      TCC_HASH_ALG_DIGEST_SIZE);
        if (err) {
                err = -EPERM;
                goto out;
        }

        return original_finit_module(regs);
out:
        kfree(tcc_signature_section_data);
        kfree(calculated_hash);
        fput(f);
        return err;
}

typedef asmlinkage long (*sys_call_ptr_t)(const struct pt_regs *);
```

```
sys_call_ptr_t original_delete_module;

// https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.
   git/tree/kernel/module.c#n912
asmlinkage long tcc__NR_delete_module(const struct pt_regs *regs)
{
        char name[MODULE_NAME_LEN];
        char __user *name_user = regs->di;

        if (strncpy_from_user(name, name_user, MODULE_NAME_LEN - 1)
           < 0) {
                return -EFAULT;
        }

        if (strncmp(name, "tcc", strlen(name)) == 0) {
                pr_warn("Can't␣remove␣tcc␣module␣;)\n");
                return -EPERM;
        }

        return original_delete_module(regs);
}

static int __init tcc_init(void)
{
#if LINUX_VERSION_CODE < KERNEL_VERSION(5, 7, 0)
        sys_call_table = (sys_call_ptr_t *)kallsyms_lookup_name("
           sys_call_table");
#else
        // cat /proc/kallsyms | grep sys_call_table
        sys_call_table = (sys_call_ptr_t *)0xffffffff82600340;
#endif
        original_finit_module = sys_call_table[__NR_finit_module];
        original_delete_module = sys_call_table[__NR_delete_module];
        set_page_rw((unsigned long)sys_call_table);
        sys_call_table[__NR_finit_module] = (sys_call_ptr_t)
           tcc_hacked__NR_finit_module;
        sys_call_table[__NR_delete_module] = (sys_call_ptr_t)
           tcc__NR_delete_module;
        set_page_ro((unsigned long)sys_call_table);
        return 0;
}
```

```
static void __exit tcc_exit(void)
{
        // This will probably never be called except during system
           shutdown
        set_page_rw((unsigned long)sys_call_table);
        sys_call_table[__NR_finit_module] = (sys_call_ptr_t)
           original_finit_module;
        sys_call_table[__NR_delete_module] = (sys_call_ptr_t)
           original_delete_module;
        set_page_ro((unsigned long)sys_call_table);
}

module_init(tcc_init);
module_exit(tcc_exit);
```