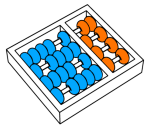
Universidade Estadual de Campinas
Instituto de Computação



Otavio Augusto Araujo Silva

Proteção de dados sensíveis através do isolamento de processos arbitrários em sistemas operacionais baseados no Linux

CAMPINAS
2016

Otavio Augusto Araujo Silva

Proteção de dados sensíveis através do isolamento de processos arbitrários em sistemas operacionais baseados no Linux

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Paulo Lício de Geus
Coorientador: Prof. Dr. André Ricardo Abed Grégio (UFPR)

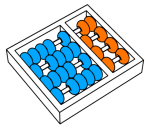
Este exemplar corresponde à versão final da Dissertação defendida por Otavio Augusto Araujo Silva e orientada pelo Prof. Dr. Paulo Lício de Geus.

CAMPINAS
2016

Na versão final, esta página será substituída pela ficha catalográfica.

De acordo com o padrão da CCPG: "Quando se tratar de Teses e Dissertações financiadas por agências de fomento, os beneficiados deverão fazer referência ao apoio recebido e inserir esta informação na ficha catalográfica, além do nome da agência, o número do processo pelo qual recebeu o auxílio."
e
"caso a tese de doutorado seja feita em Cotutela, será necessário informar na ficha catalográfica o fato, a Universidade convenente, o país e o nome do orientador."

Universidade Estadual de Campinas
Instituto de Computação



Otavio Augusto Araujo Silva

Proteção de dados sensíveis através do isolamento de processos arbitrários em sistemas operacionais baseados no Linux

Banca Examinadora:

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 26 de setembro de 2016

Resumo

Muitos problemas de segurança em sistemas Linux decorrem da forma como a segurança é gerenciada, delegando-se todas as decisões de segurança aos proprietários dos objetos (*e.g.*, arquivos ou páginas de memória). Há uma série mecanismos de segurança que visam corrigir estas lacunas no Linux ao restringir o acesso a objetos, mesmo aos seus proprietários, ou substituindo código do *kernel* por um mais seguro. Embora esses mecanismos de segurança possam evitar um grande conjunto de ataques conhecidos, eles não são destinados a evitar vazamento de privacidade de processos especiais, principalmente aquele que abusam da confiança do sistema operacional. Esta dissertação descreve a criação de um módulo do kernel Linux, com o objetivo de garantir que algumas regras sejam aplicadas em processos especias, nos quais a confiança atribuída entre os processos, pelo sistema operacional não é suficiente, e violar tais regras não é sinalizado como um ataque ou conflito de MAC(*Mandatory Access Control*). Para alcançar o objetivo proposto, interferiu-se no controle de alguns subsistemas do kernel utilizando-se de técnicas *syscall table hooking* e do LSM(Linux Security Modules) framework. Também implantou-se um simples sistema de *MAC* e comportamento suplementar para algumas chamadas de sistemas, ampliando-se assim a funcionalidade de outras abordagens (*e.g.*, AppArmor, SELinux) e principalmente o Grsecurity.

Abstract

Many issues in Linux arise from the way security is managed, by delegating security decisions to object owners(*e.g.* files or memory pages). Existing frameworks aim to remediate these issues by restricting access to kernel objects (files etc.), or by replacing the kernel code for a more secure version. Although those security frameworks may prevent a broad set of knew attacks, they do not mitigate privacy leaks, mainly those that abuse the operating system trust base. This dissertation propose a novel Linux kernel module to ensure that predefined rules are applied on sensitive processes for which the OS trust level is not enough, and neither rule breaks raise alerts or MAC(Mandatory Access Control) conflict warnings. To do so, the module takes control of a set kernel subsystems, using hooking techniques and the LSM (Linux Security Module) framework. Also it uses a simple MAC rules with an alternative syscall behavior, thus extending the functionality of other approaches (*e.g.* AppArmor, SELinux,) and especially Grsecurity.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Atualmente uma considerável parte dos *software* produzidos utiliza em alguma etapa de sua concepção conceitos de segurança, para garantir assim a integridade e determinismo de sua execução. Essa preocupação não é recente e motivou pesquisas na construção de programas mais seguros já a partir de sua compilação [68].

Tal preocupação cresceu à medida que os programas tornaram-se alvos de ataques com motivação financeira, já que um grande volume de informações valiosas são manipuladas diariamente por aqueles. À medida que computação moveu-se para fora das estações de trabalho, com o advento da computação em nuvem [38], essas informações foram migrando cada vez mais para ambientes distribuídos, onde os programas em execução lidam com dados privados, mesmo que o ambiente e seus recursos sejam compartilhados.

Entretanto, por mais que um sistema esteja preparado para resistir contra tentativas de violar a segurança, com programas e políticas de segurança coerentes a este fim, existem brechas na forma da interação entre processos que podem comprometer premissas especiais de segurança.

Essas premissas podem ser requisitos de segurança que se baseiam em critérios não presentes para todos os processos do sistema operacional, tais como a necessidade da volatilidade dos arquivos manipulados, o não compartilhamento das páginas de memória ou a restrição na troca de mensagens entre processos. Esses critérios são usualmente descumpridos pela necessidade de eficiência e desempenho dos sistemas operacionais modernos, que visam a redução de espaço em memória e disco utilizados, assim como a redução da comunicação entre processos. Tais necessidades levam a uma grande visibilidade e confiança que várias instâncias dentro do sistema operacional compartilham, principalmente aquelas de um mesmo usuário ou as instâncias de um mesmo processo (pai e filho).

O projeto dos sistemas operacionais modernos visa muito mais que fornecer uma interface simples ao usuário para interagir com o *hardware*. Esses sistemas visam fornecer tal interface para diversos usuários simultaneamente, fornecendo mecanismos e estruturas para o compartilhamento dos recursos limitados do sistema. Esse compartilhamento ocorre de tal maneira que os usuários, na forma de seus processos, interagem com os recursos acreditando no uso exclusivo deles, ou seja, os processos utilizam os recursos do sistema sem necessariamente tomarem conhecimento da concorrência envolvida nesse acesso.

Essa abstração na concorrência leva diversos processos, de vários usuários, a manterem

instâncias de um mesmo recurso, como um arquivo em disco, em seus contextos de uso. Mediante a grande necessidade de otimização, minimização do uso e alocação de recursos no sistema operacional, esses contextos por diversas vezes são explicitamente compartilhados entre os processos, principalmente aqueles de um mesmo usuário. Isto fornece-lhes a opção de acessar um recurso que um outro processo tenha adquirido, o qual podem ser dados resultantes da execução do processo e assim esse compartilhamento também ocorre como uma troca de mensagens entre processos.

O compartilhamento explícito de recursos ocorre de duas maneiras: através do compartilhamento de áreas de memória, onde um processo acessa a memória mapeada de um outro processo, podendo inclusive escrever nessas áreas compartilhadas; e através do compartilhamento de arquivos, seja através de seus descritores ou de seu caminho no sistema de arquivos. O compartilhamento de memória pode ocorrer em regiões previamente delimitadas, como as alocadas para bibliotecas compartilhadas ou aquelas utilizadas em *Inter Process Communication* [78], que são delimitadas em tempo de execução.

Em contraponto ao compartilhamento, todos os processos de um sistema operacional, como o *Linux*, têm sua memória virtualmente isolada de outros processos. Isso ocorre com o uso do espaço de endereço virtual [79] e faz com que os endereços de memória de um processo A sejam os mesmos visíveis para um processo B, apesar deles serem convertidos para endereços físicos diferentes. Esse mecanismo fornece aos processos toda a memória possível de ser mapeada pelo sistema operacional, além de evitar que um processo consiga acessar diretamente a memória de um outro processo, já que ambos usam os mesmos endereços virtuais. Entretanto é possível que processos de um mesmo usuário ou processos privilegiados[1], acessem diretamente as áreas de memórias mapeadas entre si, através de recursos como a `syscall ptrace` [31].

Ambas as formas de acesso, compartilhada e direta, só são permitidas pelo sistema operacional por existir uma relação de confiança ou de dependência entre os processos. Essa relação de dependência pode ser inerente ao processo e não essencial, como o compartilhamento das variáveis de ambiente, ou inerente e essencial, como o uso de bibliotecas dinâmicas compartilhadas. Já a relação de confiança entre processos de um mesmo usuário é inerente ao projeto do sistema operacional, onde o usuário tem a posse dos processos e os recursos por esses criados. Assim o acesso a esses recursos deve ser compartilhado o máximo possível, reduzindo o uso da memória e a intervenção direta do sistema operacional, que levaria a troca de contexto no transporte desses recursos de uma região de memória à outra. Por fim, o acesso do superusuário segue a política segundo a qual ele tem a posse do sistema e, portanto, age como dono de todos os processos.

Sendo o *Linux* um sistema operacional de uso geral, tais características do compartilhamento de recursos estão presentes, mesmo em alguns casos violando políticas de segurança que possam ser atribuídas a processos especiais. Essas políticas podem exigir, por exemplo, que as instâncias de um *webserver* executadas por um mesmo usuário, não consigam acessar regiões de memória alocadas por outras instâncias, mesmo que isso resulte em perda de desempenho ou maior uso de memória.

Tendo em vista a existência de interações baseadas na confiança entre processos dentro

[1]Nesse trabalho é considerado processo privilegiado todo aquele que executa uma instância com o *kernel*, utilizando *User Id 0* ou superusuário.

de um sistema operacional moderno, é possível que um programa que não possua falhas exploráveis seja alvo de vazamentos de dados por intermédio de ataques a sua privacidade. Esses ataques podem consistir no acesso a recursos compartilhados, como os arquivos pertencentes a um mesmo usuário ou mesmo aos endereços de memória em que estão mapeadas bibliotecas compartilhadas. Exemplos desse abuso de confiança podem ser vistos em ataques como:

- *Sudo CVE-2015-5602* [77]: através do uso de expressões regulares no arquivo de configuração (*/etc/sudoers*), um processo de um usuário contido nesse arquivo pode editar qualquer arquivo accessível pelo *sudo*, através do uso de *symbolic link*. O *sudo* tem permissões de super usuário, estendendo assim a confiança de permissões especiais sobre um diretório e assim para todo o sistema de arquivos.

- *ASLR*[2] *Mitigation*: é possível um processo descobrir quais áreas de memória um outro processo, de um mesmo usuário, está utilizando. Essa informação é obtida através da interface do *procfs* no *kernel Linux*, o que reduz a complexidade de técnicas avançadas, como em [22]. Essa redução ocorre em ataques direcionados ao *ASLR* de apenas um processo, de onde a informação é retirada.

Dessa forma, além de se proteger das falhas em *software*, alguns processos necessitam de privacidade em seus dados, tanto em disco como na memória, superior àquela que a maioria dos sistemas operacionais fornece. Essa privacidade resulta no isolamento desses processos em relação àqueles em que o sistema operacional atribuiu uma relação de confiança e dependência, pois dado que um deles seja comprometido por uma falha ou ataque, todos desta relação também o serão. Logo, esse isolamento é necessário pelas características intrínsecas desses processos: manipulam dados que deveriam ser privados a seus contextos, independentemente do usuário e de outros processos que compõem seu ambiente.

Instâncias desse problema podem ser atualmente resolvidas no ambiente Linux, através do uso de mecanismos de segurança, os quais restringem o que um processo pode acessar ou, genericamente, quais operações ele pode efetuar em um objeto. Esse objeto pode ser um arquivo, na memória ou no sistema de arquivos ou mesmo *kernel capabilities*, como a capacidade de interagir com *sockets*. Essas restrições, que compõem um sistema de *MAC*[3], são aplicadas pelos mecanismos de segurança, tais como *SELinux*, *AppArmor* e *GrSecurity*.

Esses três mecanismos de segurança foram estudados no decorrer deste trabalho, onde suas vantagens, limitações e falhas foram devidamente apontadas. Outras soluções de segurança em isolamento foram investigadas, como a virtualização completa e parcial de processos. Além das soluções atualmente disponíveis para uso, foram publicadas nos últimos anos pesquisas que resultaram em soluções capazes de resolver algumas instâncias do problema de privacidade dos dados de um processo. Essas instâncias estão relacionadas à forma em que determinados processos podem perder a privacidade, assim fazem uso de

[2]Address Space Layout Randomization
[3]Mandatory Access Control

tecnologias como: i) cifragem de páginas de memória e arquivos dos processos no armazenamento; ii) restrições na comunicação entre processos; e iii) virtualização de processos em nível de *kernel*.

Além de todas as soluções no estado da arte em isolamento e contenção de processos, também foram pesquisadas as principais falhas e mecanismos de se explorar processos, incluindo quando estão sob proteção dos mecanismos de segurança citados. Isso foi feito com o intuito de propor um sistema que fosse capaz de não agir como os atuais, mas de incrementar a proteção existente em cenários conhecidos ou mesmo naqueles que ainda não foram abusados.

Dessa forma, esse trabalho consistiu na criação de um mecanismo capaz de isolar um processo arbitrário do contexto de outros processos para aumentar a sua segurança e privacidade, mesmo no caso de processos que compartilham os mesmos privilégios. O objetivo foi complementar outros mecanismos de segurança, tais como o *AppArmor* ou o *Grsecurity*, cobrindo algumas de suas limitações para aumentar assim a segurança e privacidade, em casos onde aqueles mecanismos de segurança não consideram como possíveis de ataques. O *Grsecurity* também foi utilizado como base da proteção ao módulo proposto, uma vez que esse mecanismo é capaz de proteger o nível de *kernel* contra diversos tipos de ataques. Assim o uso do *Grsecurity*, apesar de não essencial, foi feito ao longo do trabalho como requisito de segurança.

O mecanismo proposto foi desenvolvido como um módulo do *kernel Linux, e um gerente em nível de usuário, para que certos processos façam uso de recursos especiais de segurança introduzidas no kernel pelo referido módulo.* Este módulo tem a intenção de segregar os processos em grupos, de modo a aplicar premissa especial sobre eles: cada objeto manipulado por um processo é exclusivamente da sua posse ou compartilhado entre um grupo pré-definido de processos. Para isso, o módulo de *kernel* fará uso principalmente dos seguintes recursos de *kernel*: *syscall hooking*; *LSM*[4] *framework*; cgroups (control group); CryptAPI. Além das estruturas de *kernel* citadas, o módulo também inclui as seguintes estruturas lógicas: lista completa dos processos controlados; uma lista global das estruturas de *kernel* controladas; lista de símbolos visíveis do *kernel* e máscaras de bits, por processo; mapa de regras e permissões globais, referidos pela máscara de permissões.

A utilização das referidas estruturas permite que o módulo de *kernel* controle o acesso e as permissões de cada processo ou grupo de processos, onde as permissões são compartilhadas. O controle será realizado por estruturas e símbolos do *kernel*, cujo comportamento será alterado pelo módulo, de acordo com as políticas de acesso da tupla processo-objeto. As técnicas utilizadas para interceptar estruturas e símbolos do *kernel* incluem o acesso direto aos subsistemas e o uso de técnicas de *hooking*. Diferentemente de outros sistemas de segurança no Linux, o controle de recursos proposto não utilizará os locais dos objetos, como o caminho no sistema de arquivos, mas as posses dos objetos presentes nas estruturas do módulo. Para conseguir isso, o módulo decidirá se e como, um recurso será compartilhado de acordo com a máscara de permissões, criada no primeiro processo que obteve a propriedade do recurso.

De forma resumida, as maiores contribuições deste trabalho são:

[4]Linux Security Module

- Criação de uma camada de análise na borda entre *kernel* e o nível de usuário, capaz de detectar ataques e comportamentos, em posição anterior aos sistemas que se utilizam do *LSM*;
- Mecanismo de controle pela tabela de *syscall*, possibilitando o rastreio e interação dos objetos manipulados pelos processos através da abstração do *kernel*, possibilitando a mudança de comportamento de uma mesma *sycall* para diferentes processos.
- Capacidade de evitar ou reduzir, o impacto de ataques não conhecidos através do uso de cripto-sistemas nos objetos manipulados pelo processo. Essa metodologia permite manter os dados de um processo seguro até mesmo contra as tentativas de violação do super usuário.
- Um protótipo para testes comparativos com outros mecanismos de segurança, onde o modelo proposto apresentou capacidade de barrar certas classes de ataques sem necessidade de configuração específica para cada um das formas de ataque, em que os demais subsistemas necessitavam de configuração.
- Capacidade do protótipo de barrar ataques ainda não considerados como ameaça pelos sistemas comparados.

O restante da dissertação está dividade e organizada da seguinte forma:

Capítulo 2: descreve os principais tipos de ataques realizados contra *software*, para elevar o nível de privilégio da execução ou do ambiente dos processos, obtendo assim privilégios em outros programa. Será apresentado também o modo como estes ataques são realizados e quais as técnicas usadas pelos atacantes para evadir alguns dos mecanismos de segurança estudados.

Capítulo 3: este capítulo apresentará os sistemas de proteção pesquisados durante os estudos do mestrado, capazes de proteger o *Linux* contra a maioria dos ataques existentes. Foram estudados os sistemas largamente utilizados na indústria e em ambiente *desktop*, sistemas conceituais propostos pela academia e sistemas operacionais dedicados a segurança de processos e privacidade dos usuários.

Capítulo 4: apresenta o mecanismo de isolamento de processos arbitrários a partir do contexto de outros processos, para assim aumentar a segurança e privacidade mesmo no caso de processos que compartilham os mesmos privilégios. Essa apresentação inclui a arquitetura, decisões de implementação, abordagem de proteção e limitações do sistema desenvolvido.

Capítulo 5: neste capítulo são apresentados os testes empíricos realizados para comparar o sistema desenvolvido com os outros sistemas estudados. Os testes foram apresentados de acordo com a eficiência do mecanismo proposto, suas falhas e limitações. Por fim foi testada a estabilidade do sistema, após a inclusão da proteção do sistema desenvolvido.

Capítulo 6: neste capítulo são apresentadas as conclusões e considerações finais, bem como os trabalhos futuros que podem ser derivados deste.

Capítulo 2

Ataques

Neste capítulo são descritos os principais tipos de ataques realizados contra *software*, para elevar o nível de privilégio da execução ou do ambiente dos processos, obtendo assim privilégios em outros programas. Será apresentado também o modo como estes ataques são realizados e quais as técnicas usadas pelos atacantes para evadir alguns dos mecanismos de segurança estudados.

Antes de apresentar os ataques aos programas ou ao ambiente em que eles executam, deve-se deixar claro o que é considerado uma falha ou apenas negação a um serviço. É dito que um serviço é correto quando este cumpre com sua especificação, refletida na implementação do *software*. Um defeito ou negação de serviço é o desvio do serviço fornecido, perante um evento qualquer, em relação ao serviço correto. Esse desvio, chamado de erro, significa que ao menos um estado do *software* desvia daquele dito como correto. A causa real ou suposta do erro é chamada de falha, a qual é ativa se causa um erro ou dormente caso contrário. Uma falha fica dormente até que um evento ou gatilho ocorra, o qual pode fazer parte de algum estado do programa, como uma entrada ou um evento exterior, *e.g.*, uma interrupção pelo sistema operacional. Uma visão profunda sobre os conceitos e definições da engenharia de *software* acerca das falhas, defeitos e serviços, pode ser encontrada em [49].

Apesar dos procedimentos necessários para efetuar a exploração das falhas em *software* aqui apresentadas terem sido estudados, esse trabalho não se destina a evitá-las ou mesmo consertá-las, mas sim, ao isolar o processo comprometido, evitar mais danos a outros programas ou ao ambiente. Parte desses ataques podem ser detectados no decorrer de seu fluxo de execução, porém o gatilho do ataque não será bloqueado. Isso se dá à particularidade de que alguns ataques ocorrem em programas gerados por linguagens que fornecem um baixo nível de abstração com o processador e memória. Esse baixo nível de abstração permite que estes programas possam escrever livre e diretamente em quaisquer endereços de memória mapeados para o próprio processo em questão. Logo, o sistema proposto irá atuar em conjunto com os mecanismos existentes capazes de bloquear esses ataques particulares, a fim de se também bloquear os ataques cujos gatilhos ainda não sejam conhecidos, mas o resultado do ataque possa ser identificado como tal.

Por fim, este capítulo irá listar as brechas na confiança entre processos, o que pode levar à extração de informações valiosas e que não necessariamente é considerada um ataque pelos *frameworks* de segurança existentes. Essa extração de informação pode por si só não

representar uma ameaça à integridade do sistema ou ao processo em questão, porém será visto que estas mesmas informações podem facilitar outros ataques que comprometem, ao menos, o fluxo de execução de um processo.

2.1 Execução de código arbitrário

Esta seção descreverá os principais meios de se subverter a execução de um processo, fazendo-o executar um conjunto de instruções que não estão presentes na linha de execução correta do programa. Todas as falhas aqui descritas são dependentes da arquitetura, desta forma todo o conteúdo aqui descrito foi direcionado para a arquitetura Intel x86. Além disto, esta seção trata apenas das falhas presentes em nível de usuário, que é o escopo pretendido para a solução proposta.

2.1.1 Organização da memória

As áreas de memória exclusivamente pertencentes a um processo são divididas em quatro regiões: *text* (código), *stack* (pilha), *bss* e *heap*. A área *text* contém porções do código do executável, visto que o *loader* do *kernel* os carrega sob demanda. Além de código executável, esta área contém dados de somente leitura, *e.g.*, variáveis *const* na linguagem C, quando compilado pelo `GCC`[1]. Além dessas regiões, uma quinta região pertencente às bibliotecas compartilhadas é mapeada para o processo, entre as regiões de *stack* e *heap*.

Quando o arquivo ELF[2] é executado, o código e os dois segmentos de dados são carregados em áreas separadas da memória virtual. Por convenção, o código ocupa os endereços mais baixos da memória com os dados acima e, em geral, o segmento de código é leitura-execução, enquanto o segmento de dados é leitura e escrita. Um mapa típico da memória de um processo é ilustrado na Figura 2.1.

A região de dados é dividida entre antes e durante a execução de um processo. Antes da execução esta região é dividida entre as variáveis que foram declaradas e inicializadas explicitamente pelo programador e as que não foram inicializadas. As primeiras são as variáveis globais e estáticas utilizadas pelo processo e as segundas são todas aquelas, globais e estáticas, não inicializadas. A designação para a região *bss* é dependente do compilador, pois alguns compiladores inicializam variáveis não inicializadas pelo programador como zero, e as designa ao *bss*, enquanto outros também designam as estaticamente inicializas, como zero ou *null*. As diretivas do *GCC* para designação das seções, incluindo o *bss*, podem ser encontradas em [13]. De qualquer modo, o *heap* inicia-se no topo da região *bss* e é no *heap* que todas as variáveis dinâmicas são alocadas e, diferentemente do segmento anterior, o *heap* está presente apenas na execução do processo.

O *heap* deveria armazenar somente dados resultantes da execução e nunca código executável. Entretanto alguns aplicativos utilizam a geração dinâmica de código executável, com o intuito de otimizar a própria execução, armazenando assim o código como dado no *heap* para posterior execução.

[1] GNU C Compiler

[2] *Executable and Linking Format*: padrão de arquivo executável para Linux.

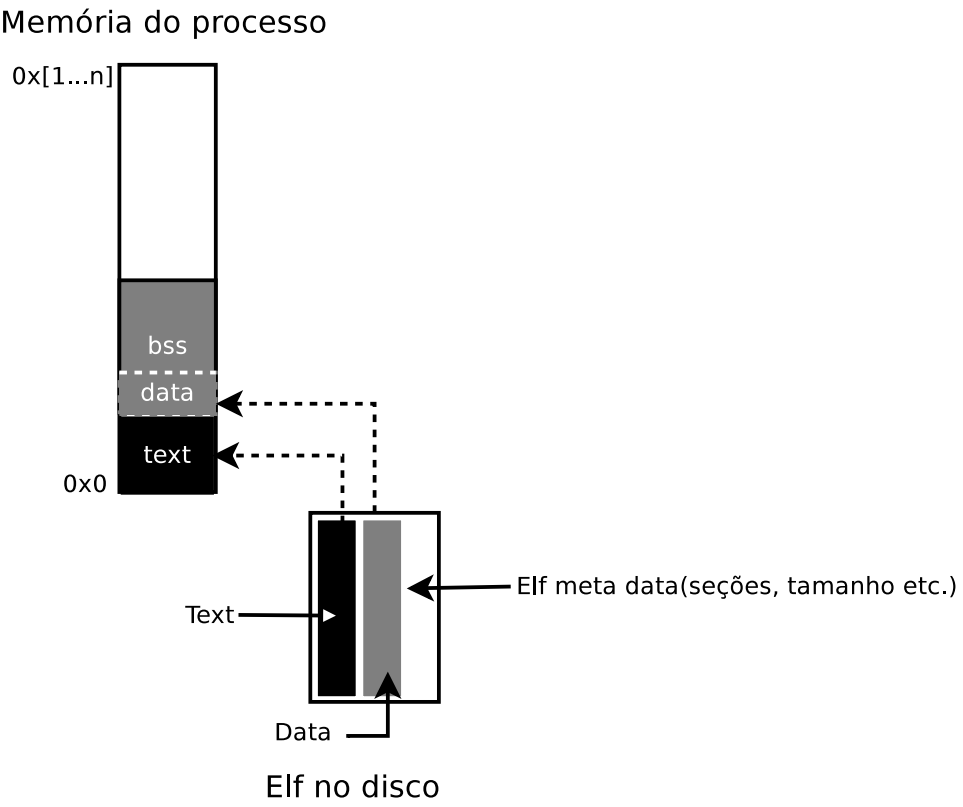


Figura 2.1: Organização Virtual de Memória: transição entre o ELF e o início da execução do processo.

Todos os três segmentos, de código, dados (incluindo *bss*) e o *heap*, são mapeados para memória real através da paginação. A Figura 2.2 mostra que o segmento *heap* cresce e diminui à medida que a memória é alocada e desalocada. Consequentemente, as entradas da tabela de paginação são adicionadas ou excluídas conforme o processo aloca e desaloca variáveis dinamicamente.

Programas escritos no estilo procedural (em oposição ao estilo orientado a objetos) são organizados como uma hierarquia lógica de chamadas de sub-rotinas. Em geral, cada chamada de subrotina envolve a passagem de argumentos do chamador para o receptor. Além disso, o receptor pode declarar variáveis locais temporárias. Argumentos de sub-rotinas e variáveis locais automáticas são acomodados no topo da memória virtual em uma área conhecida como o segmento de pilha ou simplesmente como a pilha (*stack*).

A região da pilha pode ser vista como uma região transiente de execução, com o escopo da memória limitado e local ao estado de execução corrente do processo. Programas escritos no paradigma procedimental estão organizados como uma hierarquia lógica de chamadas de sub-rotinas. Em geral, cada chamada de sub-rotina envolve a passagem de argumentos. Desta forma a pilha é uma região de memória logicamente contínua utilizada para armazenar as variáveis locais, os parâmetros a serem passados e retornos das sub-rotinas, através do uso de funções.

Esta região é mais que uma região para armazenar dados, visto que sua política de acesso é estritamente relacionada ao paradigma procedimental, assim regulando a execução de um processo, mesmo em linhas de execuções concorrentes. Consequentemente, esta

região utiliza a política *FIFO* para regular o controle do fluxo de execução do processo, onde cada novo escopo é um *frame* empilhado na pilha e o fim de um escopo retorna para o escopo anterior, o que é feito pelo desempilhamento do *frame*. Cada *frame* contém estruturas para delimitar endereços da base e topo da pilha, além do deslocamento do topo da pilha ao escopo atual ou um apontador do *frame* atual na execução, podendo assim endereçar as variáveis locais e chamas de funções aninhadas. Esse controle é feito, na arquitetura da Intel, respectivamente pelos registradores *EBP (Extended Base Pointer)* e *ESP (Extended Stack Pointer)*.

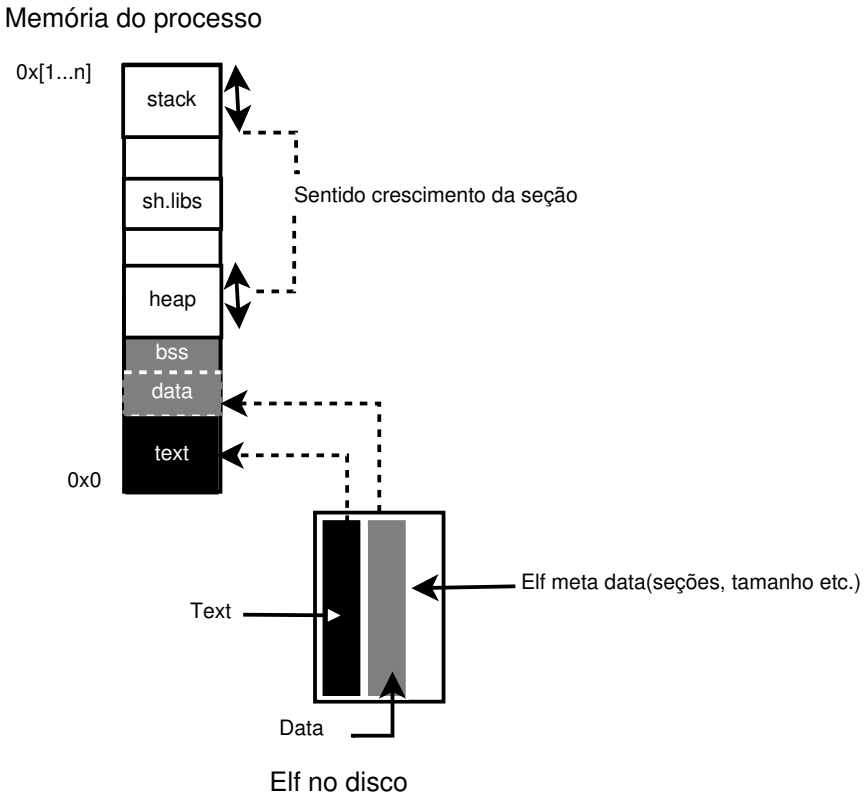


Figura 2.2: Organização Virtual de Memória: em execução, as regiões de *heap* e *stack* são criadas, e as bibliotecas dinâmicas são mapeadas no contexto do processo..

2.1.2 Buffer overflow

Um *buffer* é uma região delimitada de memória, mapeada para um processo, destinada para que este armazene dados. Essa região pode ser definida tanto antes, no início ou dinamicamente durante a execução daquele processo. O mesmo vale para a atribuição dos dados a serem definidos nessas regiões de memória. O escopo de um *buffer* pode ser definido por duas premissas: seu limite ou tamanho e a região de memória do processo na qual ele se encontra.

Sendo assim, um *buffer overflow* é uma anomalia na execução de um processo, na qual é erroneamente instruída a escrita de dados em um *buffer*, de maneira a ultrapassar os limites dele, ao ponto de sobrescrever alguma área de memória adjacente. Esta anomalia pode ser utilizada para abusar do comportamento do processo ou mesmo para tomar con-

trole do fluxo de execução deste processo e executar códigos arbitrários com as permissões do usuário ao qual o processo pertence.

Esse estouro ou *overflow* de um *buffer* é um caso especial da violação de segurança na memória, que ocorre pela falta de garantia, seja em tempo de compilação ou de execução, que um dado *buffer* não será acessado fora de seu limite ou escopo. Esses estouros estão presentes em *software* gerados por linguagens que fornecem um baixo nível de abstração com o processador e memória. Isto permite que esses programas manipulem diretamente a memória e suas estruturas de controle, assim como o processador, por meio de alguns registradores.

Desta forma, um processo consegue sobrescrever regiões linearmente adjacentes à sua própria memória, sem a princípio sofrer qualquer restrição. Outro exemplo desse controle são os programas escritos na linguagem C poderem controlar diretamente a maioria dos registradores do processador, o que permite que uma *thread* deste programa consiga manipular diretamente o *instruction pointer (IP)* e assim ser capaz de desviar o ciclo de execução previsto pelo programa.

Ataque de buffer overflow

Tendo em vista a organização da memória virtual de um processo e das condições que levam a um estouro de *buffer*, pode-se agora analisar as implicações que este estouro ocasiona na integridade e segurança de uma aplicação em execução. Como visto, o uso de linguagens de baixo nível não provê nativamente mecanismos automáticos para a checagem e validação do escopo de um *buffer*, tanto na escrita quanto na leitura. Desta forma, não há nada que impeça uma aplicação de escrever em endereços de memória arbitrários, onde esta possui permissão de escrita. A escrita pode não somente sobrescrever dados adjacentes ao *buffer* destino, como também outros *buffers* e variáveis da aplicação, estruturas de controle do fluxo de execução da aplicação e endereços de funções locais ou de retorno das bibliotecas compartilhadas. Isto possibilita que a execução da aplicação seja, de certa forma, controlada em um ataque.

Este ataque, de sobrescrita e estouro de *buffer*, pode ocorrer em todas as seções de memória onde é possível a escrita: *stack*, *heap* e *data+bss*. Consequentemente, o ataque possui diversos gatilhos particulares às respectivas seções atacadas porém, qualquer que seja a seção atacada, pode-se genericamente dividir o ataque nas seguintes maneiras:

- Sobrescrita de variáveis/*buffer*: nesse ataque, um *buffer* é sobrescrito até que se atinja o final de um outro *buffer* ou variável, possibilitando a manipulação do fluxo da aplicação, dentro dos estados previsíveis de execução. Um exemplo de ataque é a sobrescrita de uma variável de controle, *e.g.*, *isAdmin*, para forçar uma condição favorável, como *true* em *isAdmin*.

- Sobrescrita de endereço de funções internas: similar ao ataque anterior, este sobrescreve um *buffer* até atingir o final do endereço de chamada de alguma função interna, permitindo assim manipular esta chamada para executar uma outra função ou código em memória.

- Sobrescrita até alguma estrutura de controle: neste ataque uma estrutura de controle, como um endereço de retorno de uma função ou o endereço de chamada de uma função de biblioteca, é sobrescrita com o intuito de se "sequestrar" o *IP (Instruction Pointer)* e assim executar código arbitrário na memória. Este ataque diferencia-se do anterior pelo limite de estouro do *buffer*, limitado por código essencial ao processo, adjacente ao *buffer*. Se o estouro sobrescrever esse código, a aplicação terminará subitamente e o ataque falhará.

Desvio do fluxo de execução

Existem diversas maneiras de se desviar o fluxo de execução de um software, através da sobrescrita de endereços de memória. Todas essas técnicas utilizam-se de apontadores que condicionam o fluxo da execução, ou seja, apontam para estruturas em memória que, quando respeitadas e não violadas, mantêm a execução prevista do programa. Tais estruturas podem ser apontadores para funções, apontadores de retorno de funções ou mesmo apontadores na lista duplamente encadeada dos *chunks* no *heap*, os quais regulam quais porções da memória dinâmica de um processo foram alocadas ou liberadas, seus tamanhos ou ainda se estão em uso.

Apesar das diversas técnicas de desvio de fluxo empregadas nos ataques de *buffer overflow*, todas elas têm como objetivo comum a escrita de um endereço de memória, controlada pelo atacante, onde é colocado um conjunto de instruções ao processador. Como o comportamento esperado do programa é seguir ou saltar para aquele endereço, quaisquer que sejam a técnica e os meios de se chegar até a escrita do endereço, o resultado será o mesmo. Como este trabalho não se destinou a barrar especificamente este tipo de ataque, será aqui demonstrada apenas uma das técnicas, a sobrescrita do endereço de retorno ou chamada de uma função do programa. Esta técnica servirá como um demonstrativo do potencial de danos do ataque e não como uma reprodução tecnicamente profunda de um ataque de *buffer overflow* real.

Como visto anteriormente, cada chamada de função empilha um *frame* na *stack*, o qual deverá conter um endereço referente ao contexto anterior à chamada da função, o endereço de retorno. Desta forma é possível que, ao término do atual contexto, esse endereço possa ser recuperado ao desempilhar o *frame* da *stack* e assim o fluxo da execução retorne logo após a chamada da função. Logo, ao alterar-se o endereço de retorno no ataque, o contexto da execução poderá saltar para qualquer posição de memória mapeada ao processo.

A Figura 2.3 ilustra como um endereço de retorno é sobrescrito quando o *buffer* estourado se encontra na *stack*:

O contexto de execução, ou *frame*, é referenciado pelo *FP (Frame Pointer)* e pela base da pilha; na arquitetura *Intel x86* essas referências são apontadas pelos registradores *Extended Stack Pointer (ESP)* e *Extended Stack Base (ESB)*, respectivamente.

Supondo que "Buffer-alvo" seja controlado pelo atacante e seja grande o suficiente, seriam necessários cerca de *B+X+8* bytes para a sobrescrita do endereço de retorno da função. Entretanto, existem duas sutilezas neste cálculo: o valor de B será sempre o teto, em base dois, do tamanho do *buffer* definido pelo programador e o tamanho de X pode não ser conhecido ou mesmo dependente do estado da execução do processo. Desta forma,

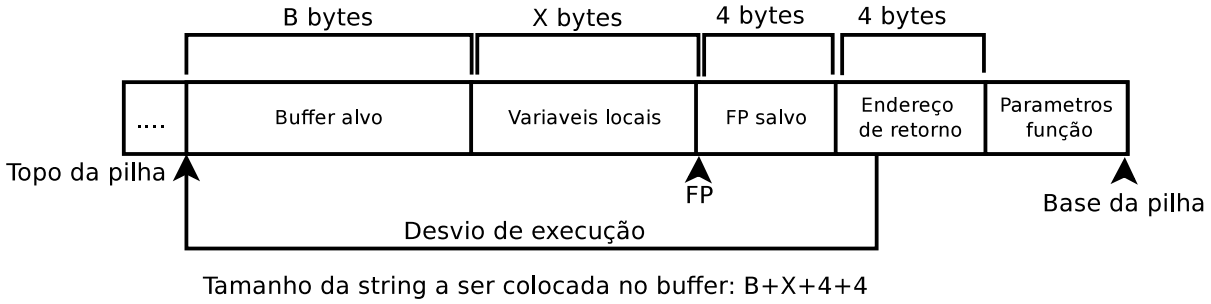


Figura 2.3: Sobrescrita de retorno baseado na *stack*.

usualmente o ataque envolve um conjunto de tentativas e erros, estimando o tamanho de X até atingir o alinhamento da *stack* do endereço de retorno. Esse palpite envolve inserções de palavras do processador contendo o valor 0x90, referente à instrução *Null Operation*. O uso desta instrução é útil, pois ela não altera o estado do processador; assim, este irá ler sucessivos *NOPs* até chegar ao endereço de retorno, que foi escrito para apontar para o início do *buffer*, que contém um conjunto de instruções úteis ao ataque, comumente nomeado de *shellcode*. Quando o fluxo de execução saltar para o início do *buffer*, o *shellcode* será executado, levando então à execução de código arbitrário no contexto da aplicação.

De forma similar ao feito na *stack*, é possível no *heap* alterar endereços pelos apontadores que são seguidos como consequência do resultado de execuções. Alguns desses apontadores são definidos em tempo de compilação, entretanto a maioria é criada na execução, como aqueles que apontam para estruturas de controle no gerenciamento do *heap*. Em qualquer dos casos, é possível estourar um *buffer* no *heap* de forma a sobrescrevê-lo. Apesar de ser comum referenciar o ataque no *bss* também como *heap-based overflow*, existem diferenças em relação a como ele ocorre. A principal diferença é que, como a seção *bss* contém dados não inicializados que crescem em direção ao *heap*, é possível sobrescrever dados contidos no *bss* em direção ao *heap*, efetuando um ataque nesse segmento. Existem também ataques de *buffers* no *bss*, que sobrescrevem variáveis globais como *isUserSU*, influenciando no controle de fluxo de tal maneira a atingir estados específicos dentre os possíveis na execução.

Existem diversas técnicas de *buffer overflow* mais sofisticadas, tais como *Format Strings* [65] e *Integer Overflow* [24], que são capazes de sobrepor certas restrições de tamanho de entrada que o programador tenha colocado justamente para evitar alguns tipos de ataques *buffer overflow*. Usando *Format Strings* é ainda possível ler áreas de memória do processo, principalmente na *stack*, e trocar apontadores dos *standard input handlers* arbitrariamente, possibilitando inserir dados em *buffers* que nem deveriam estar acessíveis pela entrada padrão.

Proteções contra buffer overflow

Os ataques modernos de *buffer overflow* envolvem uma série de técnicas para a execução arbitrária de código, aumentando a sofisticação para a evasão nos sistemas de proteção

contra *overflow*. Técnicas como a demonstrada em [9], que abusa da granularidade dos alocadores de memória dinâmicas nos sistemas operacionais para efetuar um ataque de *Heap Spray Overflow* indetectável, evoluíram ao redor das limitações dos mecanismos de proteção.

Os mecanismos de proteção contra ataques de *buffer overflow* dividem-se em dois grupos, os que obstruem o ataque antes que ele ocorra e aqueles que o detectam durante sua execução. O primeiro grupo emprega mecanismos capazes de inviabilizar o ataque, de tal maneira que este não possa ser concluído mesmo com a existência de falhas no *software*. Já o segundo não evita que o ataque ocorra, porém utiliza meios de detectá-lo para, em seguida, tomar uma ação que evite a continuidade do ataque, *e.g.*, terminando forçosamente o processo vítima do ataque. Esse segundo grupo de mecanismos de proteção é foco dos estudos deste trabalho e será profundamente revisto nos próximos capítulos. Os mecanismos capazes de impedir o ataque antes que ele ocorra podem ser divididos em quatro tipos:

1. **Linguagens e bibliotecas seguras:** O uso de linguagens de programação que permitam um alto grau de manipulação da memória principal, incluindo suas estruturas de controle, e dos registradores do processador integram os fatores que definem a aplicabilidade dos ataques de *buffer overflow*. Assim, a escolha da linguagem a ser usada e de seu compilador pode oferecer alguma vantagem na proteção dos ataques mencionados. Esta vantagem advém dos mecanismos que checam os limites de acesso aos *buffers* e principalmente à visibilidade da memória oferecida ao programador, possibilitando assim o acesso às estruturas de dados da gerência de memória. Mesmo que a linguagem e o compilador usado não ofereçam grandes vantagens, ainda é possível impor limites e escopos ao uso de *buffer* nas linguagens de baixo nível, através do uso de bibliotecas consideradas seguras para a manipulação de memória.

   Essas bibliotecas, previamente testadas quanto à segurança, utilizam tipos abstratos de dados e controle automático e centralizado da gerência de *buffers*, incluindo o respeito aos limites e formas de acesso a seu conteúdo, podendo limitar o impacto ou mesmo evitar a ocorrência de ataques. Entretanto, tal solução repassa o ponto de falha ou ruptura para o conjunto de bibliotecas dadas como seguras, já que são elas que fazem o acesso direto à memória principal e aos registradores. Um exemplo desse tipo de falha foi o *CVE-2015-7547* [86], onde diversas funções da biblioteca *libresolv*, contida na *GNU C Library*, estavam suscetíveis a *stack buffer overflow*.

2. **Sobrescrita supervisionada:** Um dos grandes problemas destes ataques é a sobrescrita de estruturas de controle, pois assim o atacante pode controlar regiões não atingíveis da execução ou ser capaz de executar código arbitrário. Então, caso seja possível detectar que houve uma escrita nessas estruturas, seria possível interromper um ataque em andamento e assim evitar seu sucesso.

   Esse controle é feito através da adição de marcadores ou canários ao fim de cada contexto pela sua análise e comparação antes de um retorno ou chamada de uma função, assim detectando se esse endereço a ser seguido foi alterado por uma sobrescrita.

Apesar de existirem sistemas que aplicam esse conceito tanto na *stack* quanto no *heap*, a maioria o faz na *stack*. Mesmo assim, trabalhos como [43] prevêm o uso de canários no *heap* especificamente para ataques na alocação e desalocação de memória, evitando-se assim alguns impasses que o uso de canários em todas as estruturas do *heap* geraria.

Atualmente a maioria dos compiladores pode usar canários para a *stack*, tais como *GCC, Visual Studio* e *CLANG*. Existem três tipos de canários: *Terminator, Random,* e *Random XOR*. Sob *gcc*, atualmente o *StackGuard* suporta todos os três tipos, enquanto o *ProPolice* suporta apenas os tipos *Terminator* e *Random*. Os autores em [50] apresentam a abordagem adotada pelo *StackGuard*, suas qualidades e o *trade-off* no desempenho ao utilizar mecanismos de proteção na *stack*. Já os autores em [52] revisam duas décadas de mecanismos de proteção a *overflow* e propõem uma abordagem que cobre as falhas presentes nos mecanismos de canários, que seja resistente a ataques de força bruta tipo *BROP (Blind Return-Oriented Programming)* e ataques contra *ASLR(Address space layout randomization)*.

3. **Execução em memória protegida:** O ataque mais comum e poderoso é o que leva à execução arbitrária de código na memória, através do desvio do fluxo de execução do programa para que este execute instruções armazenadas como dados, na *stack* ou no *heap*. Este ataque pode ser mitigado por um recurso que marca as páginas de memória como não executáveis, evitando assim que qualquer dado armazenado lá seja executado. Com esse recurso, um ataque de *buffer-based overflow* resultaria em uma exceção ou falha de permissões.

   Esta marcação ocorre com o uso de um bit na *PTE (Page Table Entry)* na tabela de paginação da arquitetura *x86_64* e é nomeada como *Intel NX* (No eXecute) e *AMD XD* (eXecute Disabled). Na arquitetura *x86* essa proteção é feita através de emulação, com o uso de *PAE (Physical Address Extension)*, em *patches* no *kernel* como o *PaX* para Linux ou `W^X` na família BSD, que permitem desabilitar a execução de código em certas páginas de memória. Em todo caso, apesar de impedir ataques, essa proteção também evita a execução de aplicativos que dependam da execução em memória, de códigos dinamicamente gerados. Exemplos desses aplicações são facilmente encontrados, como máquinas virtuais, emuladores, contêiners de vídeo e mesmo o *Xorg* para aceleração gráfica bidimensional.

4. **Aleatorização do espaço de memória:** Uma das premissas para que os ataques nas seções de memória ocorram com sucesso é o conhecimento prévio pelo atacante de onde estarão as estruturas a serem sobrescritas ou utilizadas no ataque, tais como os endereços de bibliotecas compartilhadas ou aponteadores de funções. O uso de *Address Space Layout Randomization (ASLR)* consiste na organização pseudo-aleatória do mapeamento virtual de áreas de memória, como as posições das seções *text, stack, bss, heap* ou das bibliotecas compartilhadas de um processo. Esse mapeamento muda, ou deveria mudar, para cada processo lançado pelo sistema operacional, tornando de difícil predição tanto os endereços de memória utilizados em um ataque como o *offset* entre as regiões e alguns apontadores importantes.

Uma consequência simples disso é a impossibilidade do ataque não funcionar genericamente, ou seja, para qualquer programa vulnerável em qualquer ambiente. Em vista disso, é necessário alterar o ataque de tal forma a contornar essa variação nos endereços, tornando-o extremamente mais complexo ou inviável. A maioria dos sistemas operacionais modernos emprega o *ASLR* para os processos e algumas áreas do próprio *kernel*. Apesar de eficiente, essa tecnologia pode ser suplantada por diversas técnicas dependentes do S.O utilizado. Os autores em [22] demonstram um ataque capaz de reduzir a aleatorização, através da predição da influência dos campos pseudo-aleatórios. Outra abordagem de ataque é através do vazamento de informações do processo, como o endereço virtual de uma biblioteca mapeada, capaz de comprometer o *ASLR* por completo ou facilitar ataques como o de *brute force*.

2.2 Comprometimento na privacidade de um processo

A seção anterior listou ataques que ativamente alteram o controle de fluxo de um processo, comprometendo toda a sua execução, assim como todos os seus objetos, *e.g.*, arquivos, páginas de memória e *sockets*. Existem, entretanto, ataques com baixa ou nenhuma interferência no fluxo de execução do processo, cujo objetivo é o acesso às informações ou objetos resultantes da execução. Esses ataques usualmente utilizam um erro ou conflito de permissões, tanto na heurística do programa quanto na do sistema operacional, de maneira a obter acesso indireto a um processo, através de seus dados. Os dados obtidos podem ser utilizados como parâmetros em outros ataques, como a redução do espaço de busca em um criptossistema, identificação de diretórios ou o sistema de arquivos.

Esse tipo de ameaça leva ao vazamento de informações da execução de um processo e dos recursos disponíveis a ele (como permissões no sistema de arquivos) a outros processos do sistema. Isto permite a um processo ganhar acesso a recursos com requisitos de privacidade de outro processo, de tal maneira a agir perante tais recursos como proprietário deles. Assim, se um processo obtêm uma referência na *kernel keyring*[3] de outro processo, ele pode por exemplo conseguir a chave de um criptossistema utilizada pelo outro processo. Um exemplo mais simplório seria o uso do sistema de arquivos por dois processos de um mesmo usuário, onde cada processo executa uma instância de *virtual host* diferente do Apache.

Desta forma fica claro que, diferentemente dos ataques da seção anterior, o vazamento de informação não é necessariamente condicionado pela presença de falhas de comprometimento da linha de execução, seja no programa ou no sistema operacional. Tal forma de ameaça pode portanto ser subdividida em dois grupos, aquele com ataques que necessitam de uma falha (heurística, implementação ou *bug*) e aquele que contém ataques que abusam da confiança e privilégios do sistema operacional.

Foi mencionado anteriormente a falha do `sudo` (CVE-2015-5602) [77], onde é possível explorar o mecanismo de resolução de expressões regulares por parte do `sudo`, em conjunto com a resolução de permissões de *symbolic links*, a fim de ler arquivos arbitrários com permissões de superusuário. Essa falha ocorreu por um *bug* na heurística do `sudo`, ao

[3] Conjunto de chaves por processo, utilizadas nos criptossistemas incluídos na *CryptoAPI*

não avaliar o *path* final corretamente, através da resolução das expressões regulares e da obtenção do *endpoint* no *symbolic link*. Um outro ataque que levou ao vazamento de dados através de falha foi o da *Linux keyctl syscall*[88], que permitia a um dado processo com acesso a um objeto do *kernel keyring* obter a referência a outro objeto através de força bruta: tentava-se todas as possibilidades de apontadores, dependendo da arquitetura 32 ou 64 bits. Assim, um processo seria capaz de acessar qualquer objeto cifrado, incluindo aqueles por intermédio de módulos de segurança, sem de fato ter permissões para tal.

O exemplo mais simples de que a confiança atribuída por processos de um mesmo nível de privilégios pode ser abusada e levar ao vazamento de informações é o sistema de arquivos, haja vista que processos de um mesmo usuário podem, no mínimo, acessar os mesmos arquivos e assim abusarem da confiança dada pelo sistema operacional. Entretanto, a segurança provida a um processo pelo *kernel* ou por um *security framework* é passível de ser comprometida pela leitura das entradas específicas pertencentes a processos de um mesmo usuário no *procfs* [42].

O *procfs* é referido como um pseudo-sistema de arquivos que contém informação sobre processos e sobre o *kernel Linux*. Ele não contém arquivos em armazenamento secundário, mas arquivos virtuais apontando para regiões de memória do *kernel* correspondentes às informações do sistema sobre os processos em tempo de execução (*e.g.*, gerenciamento de memória, dispositivos montados etc.). Ele também contém entradas referentes a cada processo em execução (*i.e.*, `/proc/pid`), onde é possível encontrar informações como memória mapeada (`/proc/pid/maps`, incluindo a posição das bibliotecas compartilhadas) , arquivos abertos, dispositivos montados pelo processo, entre outras. De posse dessas informações, um processo poderia descobrir quais áreas de memória um outro processo de um mesmo usuário está utilizando, sendo capaz de reduzir ou mesmo anular a complexidade de ataques contra a *ASLR* de um determinado processo. A única maneira de evitar esse ataque é a redução da confiança entre processos pelo *kernel Linux*, de tal forma que um processo tenha acesso a informações no *procfs* pertencentes exclusivamente a ele mesmo. Essa abordagem foi introduzida ainda no *kernel 2.4* pelo *Grsecurity* [73], antes mesmo desse sistema introduzir o conceito de *ASLR* no *Linux*.

Além do uso do *procfs*, existem outras maneiras de ter acesso às informações do *kernel* sobre um processo no Linux, de forma a abusar da confiança do sistema operacional, desde o uso de *debuggers*, como o *GDB* [55], ao uso de *process tracing*, através da *syscall ptrace* [31]. Com o uso do *ptrace*, um usuário é capaz de examinar a memória e estado de execução de qualquer de seus processos. Assim, caso um processo seja comprometido, seria possível a um atacante anexar-se a outros processos em execução, extraindo informações da memória ou mesmo injetando código para execução. Um utilitário para Linux, o *Linux Injector* [74], faz exatamente isso: injeta e executa instruções na memória de qualquer processo do mesmo usuário.

Outro exemplo de abuso da confiança do sistema operacional, pelo mau uso das funções de um *device driver* por um processo, está presento em [87]. Possivelmente devido a retrocompatibilidade, um *device driver* da Asus possuía permissões relaxadas, pelo *kernel Windows*, para acesso ao *ring0* através de usuários não privilegiados. Esse acesso relaxado, em conjunto com o uso das funções de leitura e escrita em endereços físicos pelo *device driver*, permitiram que usuários de baixo privilégio manipulassem livremente a memória,

incluindo a memória do *kernel*. A resolução desse problema foi a invalidação da assinatura de versões de tal *device driver* datados de 1993, evitando assim que fossem carregados pelo sistema.

Capítulo 3

Trabalhos relacionados

Este capítulo apresentará os sistemas de proteção relacionados, capazes de proteger o *Linux* contra a maioria dos ataques previamente demonstrados. Foram estudados os sistemas largamente utilizados na indústria e em ambiente *desktop*, sistemas conceituais propostos pela academia e sistemas operacionais dedicados a segurança de processos e privacidade dos usuários. Tais sistemas representam assim o estado da arte em contenção, isolamento e segurança entre processos dentro do *kernel* de um sistema operacional. Além dos sistemas de proteção estudados, este capítulo introduzirá os mecanismos de gerência e controle de acesso utilizados para às impor políticas de seguranças utilizadas pelos sistemas de proteção.

3.1 Modelos de controle de acesso

Todo sistema de segurança cuja função seja isolar e conter processos, em escopos ou níveis de permissões diferentes, utiliza uma gerência de acesso baseada em um modelo de controle de acesso. Esse controle visa separar as ações possíveis de serem executadas por um sujeito (*i.e*, *software*) em um dado conjunto de objetos (*e.g.*, arquivos). Com essa divisão, é possível aplicar políticas de acessos baseadas na relação entre objeto-sujeito e a requerida ação, provendo assim restrições ao livre acesso dos objetos. Um modelo abstrato de controle de acesso foi proposto inicialmente como uma *ACM (Access Control Matrix)* [17], e apesar de atualmente *ACM* não ser considerado um modelo de controle de acesso, seu uso é feito por diversos dos atuais modelos de controle de acesso.

Uma *ACM* pode ser vista como uma matriz retangular de células, com uma linha por sujeito e uma coluna por objeto. A entrada em uma célula - isto é, a entrada para um determinada tupla sujeito-objeto - indica o modo de acesso que o sujeito está autorizado a exercer sobre o objeto. Cada coluna é equivalente a uma lista de controle de acesso para o objeto, e cada linha é equivalente a um perfil de acesso para o sujeito, também chamado de *ACL (Access Control List)*. Um exemplo de *ACL* pode ser encontrado na maneira como usualmente o acesso aos arquivos em um sistema operacional é gerenciado. A Tabela 3.1 representa as operações possíveis de serem realizadas pelos sujeitos (usuários), nos objetos (arquivos) do sistema de arquivos, formando por fim uma *ACM*.

O controle de acesso consiste em decidir se um sujeito que produz uma solicitação

	File1	File2
User1	read, write, execute	read
User2	read, execute	read, execute

Tabela 3.1: *Access Control Matrix*, onde cada entrada por linha, representa a relação dos usuários e as operações nos arquivos.

deve ser ou não confiável nesta mesma solicitação. Por exemplo, o sujeito pode ser um processo em execução de um determinado usuário, e o pedido pode ser um comando para ler um arquivo específico. Neste exemplo, o gerente de controle de acesso seria responsável por decidir se a leitura deveria ser permitida. Esta decisão de autorização pode, no caso mais simples, basear-se na consulta de uma *ACM* mapeie o nome do usuário e o nome do arquivo, para um conjunto de operações permitidas.

Desta forma, uma definição mais restrita de controle de acesso abrangeria apenas os requisitos envolvidos para a aprovação do acesso pelo sistema gerente das políticas (*e.g.*, sistema de segurança, sistema operacional etc.), ao basear a tomada da decisão nas as políticas vigentes, tendo como requisitos o nível de autenticação ou acesso do sujeito e no nível de autenticação ou acesso do objeto. Autenticação e controle de acesso são muitas vezes combinados em uma única operação, de modo que o acesso é aprovado com base na autenticação bem-sucedida, e na existência de uma regra ou política que permite o acesso do indivíduo autenticado ao objeto específico. Um exemplo dessa combinação está presente, *e.g.*, no acesso de uma página de memória compartilhada entre dois processos, onde a autenticação do usuário –por meio da existência do processo– é feita de maneira implícita e simultânea ao controle de acesso pelas permissões vigentes, que regulam quais páginas de memória e modo de acesso estão disponíveis

Os modelos de controle de acesso tendem a serem descritos [1] em uma de duas classes: aquelas baseadas em capacidades e aquelas baseadas em listas de controle de acesso (*ACLs*).

- Em um modelo baseado em capacidades, a posse de uma capacidade é feita através da existência de uma referência ou característica exclusiva ao sujeito e objeto. O acesso a um dado objeto é partilhado pela transmissão dessa capacidade, e qualquer sujeito que detenha tal capacidade, retem o acesso ao objeto a qual ela referencia. Um exemplo seria a posse de uma chave, ou *token*, que permite a leitura de um dado.

- Já um modelo baseado em *ACLs*, o acesso de um sujeito a um objeto depende se sua identidade aparece em uma lista associada ao objeto, contendo suas permissões. O acesso é transmitido editando-se a lista para inclusão ou remoção de sujeitos e permissões. Diferentes sistemas que utilizam *ACLs* têm uma variedade de diferentes convenções sobre o responsável pela edição da lista, assim como ela é editada.

Dentre os modelos de controle de acesso, os três mais utilizados pelos mecanismos de segurança são: *Mandatory Access Control (MAC)*[2], *Role Based Access Control (RBAC)*[11] e *Discretionary Access Control (DAC)* [60]. Além desses, existem outros modelos não

muito utilizados, tais como: *Attribute-based Access Control (ABAC)* [21], *Identity-Based Access Control (IBAC)* [35] e *Organization-Based Access control (OrBAC) [29].* O modelo *DAC* utiliza um conjunto de *ACLs*, para regular o acesso aos objetos controlados. Estas listas de permissões são compostas pela identificação do sujeito, como o identificador de usuário do sistema operacional, e um identificador do objeto, como o nome, *path*, *inode* etc. Esse modelo é o único, cujo proprietário do objeto é o responsável em determinar as politicas de acesso do objeto, para o próprio sujeito e usualmente para outros grupos de sujeitos do sistema.

Diferentemente do *DAC*, no modelo *MAC* os sujeitos não têm liberdade para determinar quem tem acesso aos seus objetos, uma vez que as regras que regem o modelo são comportas por uma outra entidade, *e.g.*, um mecanismo de segurança. Nesse modelo, os objetos podem receber etiquetas de permissão e acesso, fornecendo uma outra abstração para a gerência de segurança, uma vez que não existe mais a relação de identificação do sujeito, e sim a posso daquela etiqueta. Assim é possível regular o acesso de acordo com um conjunto de regras, para sujeitos quem já possuem o acesso cedido por um outros gerente de acesso, *e.g.*, o sistema operacional. Esse modelo é o mais largamente utilizados pela maioria dos mecanismos de segurança e controle de acesso, tanto para sistemas operacionais quanto políticas de rede e monitoramento.

Baseando-se também no uso de capacidades, o *RBAC* permite a gerência de acordo com o papel, ou posição hierárquica, de um sujeito perante o sistema, para assim regular o acesso. Desta forma, todos os sujeitos são separados de acordo com seu papel, assim como os objetos, e apenas aqueles pertencentes a mesma posição, tem acesso garantido. Os mecanismos de segurança usualmente dividem os processos de um sistema regido pelo modelo *RBAC*, em papéis (ou *roles*) tais como: *boot*, serviços do sistemas, serviços dos usuários, usuários administrativos, usuários de impressão, usuários web etc. E desta forma, de acordo com um conjunto de regras e políticas de acesso, ele permite a separação do sistema nos papeis equivalentes ao sua funcionalidade.

Apesar de não ser utilizado pelos principais mecanismos de segurança do Linux, o *ABAC* foi estudado e parcialmente utilizado no mecanismo de segurança aqui proposto, principalmente pela sua versatilidade. Este mecanismo de controle de acesso baseado em atributos, define um paradigma de controle de acesso pelo qual os direitos de acesso são concedidos aos sujeitos através do uso de políticas que combinam um conjunto de atributos. As políticas podem usar atributos como a identificação do usuário, recursos do sistema (*e.g. mount point*), o ambiente (*e.g.*, bibliotecas dinâmicas) etc. Este modelo faz o uso da lógica booleana, na qual as regras contêm instruções *if, then, else* sobre quem está fazendo a solicitação, o objeto e a ação.

Diferentemente do *RBAC* que emprega funções pré-definidas no sistema (os *roles*), as quais possuem um conjunto específico de privilégios associados, no *ABAC* o conceito de políticas é expressado pelo conjunto de regras booleanas complexas, tornando dinâmico o conceito de privilégios. Uma vez que esses atributos podem mudar para um mesmo sujeito e objeto, o acesso também pode variar em função desta mudança. Esse modelo apesar de ser considerado inovador, somente foi completamente empregado (seguindo todas as especificações), recentemente pela *eXtensible Access Control Markup Language (XACML)* [47].

3.2 Linux Kernel Security Overview

O Linux foi inicialmente desenvolvido como um clone do sistema operacional Unix, a partir de 1990. Como tal, ele herda o núcleo do modelo de segurança Unix—uma forma de *DAC (Discretionary Access Control)*. As características de segurança do *kernel Linux* têm evoluído significativamente para atender às exigências modernas, embora o *Unix DAC* permaneça como o modelo principal.

Resumidamente, o *Unix DAC* permite que o proprietário de um objeto, tal como um arquivo, defina a política de segurança desse objeto, ou seja, sob a discrição do usuário. Assim, um usuário pode, por exemplo, criar um novo arquivo no seu diretório pessoal e decidir quem mais poderá manipulá-lo. Essa política é implementada como bits de permissões de acesso armazenados no *inode* do arquivo, permissões essas que podem ser definidas separadamente para o proprietário, um grupo de usuários específico e para todos os demais usuários. Esta é uma forma relativamente simples de uma lista de controle de acesso, *ACL (Access Control List)*. Esse mesmo esquema de permissões é seguido também para processos de um usuário, ou seja, programas executados por um usuário levam todos os direitos deste, com a política de separação entre usuário proprietário e os demais.

O *Unix DAC*, sendo um esquema de segurança relativamente simples, não atende todas as necessidades de segurança em sistemas modernos, onde os usuários podem não ser confiáveis. Ele não protege adequadamente contra usuários mal intencionados, programas mal configurados ou com falhas, que podem ser explorados para o acesso a recursos não autorizados. Aplicações privilegiadas, aquelas que executam como processos do superusuário, são particularmente críticas nesse ambiente pois, uma vez comprometidas, elas podem fornecer acesso total ao sistema a um atacante.

Várias das primeiras extensões de segurança do Linux foram melhorias de funcionalidades ao esquema *Unix DAC* implementado, como a inclusão de *POSIX ACLs*[1] e *POSIX Capabilities* no *kernel*. *POSIX Access Control Lists* estende o *Unix DAC ACLs* para um esquema muito mais refinado, permitindo permissões específicas para vários usuários e grupos individualmente. A gerência das *ACLs* é feita pelos aplicativos `setfacl` e `getfacl`, que fazem a interface com os atributos estendidos do sistema de arquivos, os quais são armazenados como metadados nos *inodes*. O *Linux kernel* suporta atualmente o *POSIX 1003.1e Draft Standard 17* para *POSIX ACLs* e *Capabilities*.

POSIX Capabilities têm como objetivo granularizar o poder normalmente associado ao super-usuário, de modo que uma aplicação que exija algum privilégio elevado não tenha que receer todos os privilégios do sistema. Assim, ela é executada com um ou mais privilégios, como *CAP_NET_ADMIN* para gerenciar recursos de rede, sem possuir outros privilégios especiais para manipular o sistema como o super-usuário normalmente teria. Esses recursos são geridos, por aplicação, com o uso de `setcap` e `getcap`, possibilitando assim reduzir o número de aplicações com *User Id 0* no sistema, através da atribuição de capacidades específicas para elas. A partir da família *2.2* do *Linux kernel*, foi possível manipular a lista de *Capabilities* disponível no sistema, de tal maneira que tornou-se possível retirar *Capabilities* globalmente e assim retirando poder até do

[1] POSIX (*Portable Operating System Interface)*, é uma família de normas definidas pelo IEEE para a manutenção de compatibilidade entre sistemas operacionais e designada formalmente por IEEE 1003.

super-usuário no sistema. Essa lista está sempre completa no *boot* e pode sofrer remoções (porém nunca inclusões) até o próximo *boot*; um exemplo desse mecanismo é a remoção de *CAP_ AUDIT_ CONTROL*, impedindo que existam mecanismos de auditoria ou *debug* no *kernel*, independentemente do usuário que tentar tais ações.

Outra característica de segurança adicionada no Linux foi o uso de *namespaces*. Ela deriva do sistema operacional *Plan 9* [54] e é uma forma leve de particionar recursos do sistema, na maneira como são vistos por processos. Com o uso de *namespaces* é possível que cada processo tenha a sua própria visão dos pontos de montagem dos sistema de arquivos ou até mesmo da tabela de processos. Assim, um *namespace* envolve um recurso global do sistema em uma abstração que faz parecer, para todos os processos dentro de um mesmo *namespaces*, que eles têm sua própria instância isolada do recurso global. Alterações nos recurso globais são visíveis apenas para os processos que são membros de um mesmo *namespace* e invisíveis para outros processos. Um uso comum é na implementação de contêiner de processos.

Uma documentação detalhada sobre o uso e gerência de *namespaces* pode ser encontrada em [32]. Um grupo de processos pode conter um ou mais dos seguintes *namespaces*:

Namespace	Constante	Isola
Cgroup	CLONE_NEWCGROUP	Raiz do diretório do *Cgroup*
IPC	CLONE_NEWIPC	*System V IPC, POSIX message queues*
Network	CLONE_NEWNET	*Network devices, stacks*, portas etc
Mount	CLONE_NEWNS	Pontos de montagem dos *filesystems*
PID	CLONE_NEWPID	Process IDs (visão dos *PIDs* de outros processos)
User	CLONE_NEWUSER	User/group IDs (um processo não conhece seu user/group real)
UTS	CLONE_NEWUTS	*Hostname* e *NIS domain name*

Apesar de todos os mecanismos listados aumentarem o controle e gerência de acesso no Linux, eles não foram construídos especificamente para proverem segurança ao sistema, *e.g.*, contra ataques ao *software* ou comportamento malicioso por parte dos processos e seus usuários. Por isso, a partir da família *2.6* do *kernel*, foi introduzido o *LSM API (Linux Security Module API)*.

3.3 Linux Security Module API

O *Linux Security Module API* [40] foi projetado para fornecer um mecanismo unificado de monitoração e controle de segurança ao nível de usuário, unificando assim os *security frameworks* para o Linux. Seu projeto, que foi feito impondo o mínimo de alterações possíveis no *kernel*, permitiu que o *SELinux* fosse integrado ao *kernel vanilla*, fomentando assim outros projetos de segurança. Desta maneira, sua criação foi feita para padronizar e simplificar a introdução de sistemas de segurança ou *security frameworks* no Linux. Esta *API* utiliza *hooks* em cada ponto do *kernel*, onde uma *system call* resultará em acesso a um importante objeto de kernel (como *inodes* ou *task control blocks*), para fornecer uma interface de *callbacks* aos módulos de segurança do *kernel*.

Um módulo que utiliza essa *API* precisa registrar-se no *kernel* como um *Linux Security Module (LSM)*, para assim receber as *callbacks* a partir dos *hooks* implementados. Mais especificamente no processo de registro, o módulo precisa informar uma lista de funções que serão apontadas pelas *callbacks* e assim os respectivos *hooks* são ativados pelo *kernel*. Todas as chamadas pertinentes a segurança são transmitidas atomicamente para o módulo, evitando assim *race conditions*.

O *LSM API* permite que modelos de segurança diferentes sejam registrados no *kernel*; apesar de usualmente esses modelos serem *access control frameworks*, não há restrições para outros modelos. Para garantir a compatibilidade com as aplicações existentes, os *hooks* são colocados de modo que os controles do *Unix DAC* sejam realizados primeiro, sendo que somente se eles tiverem sucesso o código do *LSM* é invocado. Assim é possível a monitoração e controle do acesso aos objetos do *kernel* ou dos processos, para a aplicação das regras de acesso e gerência de permissões. Esses objetos controlados podem variar de arquivos, *pipes* e até *kernel capabilities*—como a capacidade de interagir com *sockets* ou executar processos como superusuário.

Neste trabalho o *AppArmor* e o *SELinux* foram profundamente estudados, por serem os dois principais *security frameworks* presentes no *kernel vanilla* que utilizam a interface do *LSM*. Eles são os mais utilizados tanto em sistemas domésticos quanto embarcados, chegando até mesmo em ambiente industrial. Porém, existem outros sistemas presentes no *kernel vanilla*, que utilizam a interface do *LSM*: *Smack, TOMOYO* e *Yama*. O *Smack* [61] propõe um sistema de *MAC (Mandatory Access Control)* simplista para ambientes embarcados com poucos recursos de *hardware*. Ele implementa um esquema de controle baseado em etiquetas para os objetos, com uma política personalizada de acesso. Já o *TOMOYO* [8] possui uma aplicação balanceada tanto em auditoria e métricas do sistema quanto em segurança. Seu projeto foi focado em simplicidade, através da utilização de um modo de aprendizagem, onde o comportamento do sistema é observado com a finalidade de gerar políticas de segurança. O *Yama* [7], diferentemente dos outros módulos, não utiliza um *MAC* para gerenciar o acesso a objetos. Sua implementação resume-se a impor restrições, por domínio de acesso, a processos quanto ao uso do *ptrace*. Ele implementa diversos mecanismos que limitam o acesso disponível ao *ptrace*, de acordo com o usuário e a hierarquia do processo. Por exemplo, supondo uma árvore de processos com a raiz no *init*, um processo de um mesmo usuário só consegue anexar-se a outro que estiver em um nível inferior a ele. Essa heurística evita que processos mais novos monitorem os mais antigos e pode ser reforçada ao nível de *children-only logic*, inicialmente proposto e usado pelo *Grsecurity*, que limita a interação apenas para pai-filho.

3.3.1 SELinux

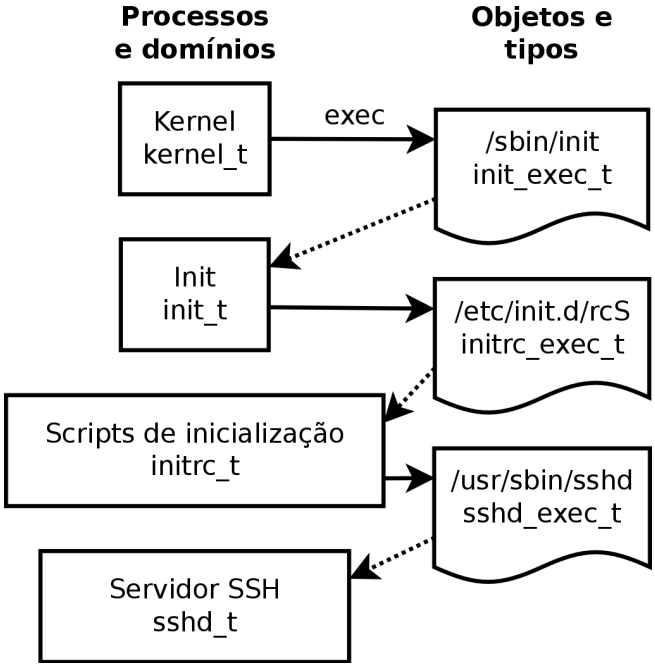
SELinux (Security Enhanced Linux) [41] é uma implementação de *MAC* [76], projetado para atender a uma ampla gama de requisitos de segurança, do uso doméstico, passando pelo industrial e chegando até governamental e militar. A gerência de acesso por *MAC* difere daquela utilizada por *DAC*, pois a política de segurança e acesso é administrada centralmente, assim os usuários não administram a política dos seus próprios recursos. Essa abordagem ajuda a conter ataques que exploram falhas em *software*, usuários mal

intencionados ou configurações incorretas.

Esse *security framework* segue o modelo do menor privilégio necessário. Em um cenário de práticas rigorosas, tudo é negado por padrão e, somente após a escrita de uma série de exceções nas políticas, libera-se o estritamente necessário para um processo funcionar. Tais políticas são escritas especificamente para cada executável cujos processos se deseja controlar. Para tal controle, todos os objetos do sistema, como arquivos, processos e diretórios, recebem *labels* com informações de segurança, o que é chamado de *SELinux context*. Um exemplo dessa etiqueta pode ser visto a seguir:

```
~]$ ls -Z file1
-rw-rw-r--  user1 group1 unconfined_u:object_r:user_home_t:s0 file1
```

Neste exemplo, o arquivo *file1* é identificado pelo seu *inode* e o *label* fornece as seguintes informações: o usuário *unconfined_u*, o *role object_r*, um tipo *user_home_t* e um domínio *s0*. Estas informações são usadas para tomar as decisões do controle de acesso no *SELinux*. Por padrão, os arquivos e diretórios recém criados herdam o tipo do *label* de seu diretório pai. Por exemplo, ao criar um novo arquivo no diretório */etc*, que é rotulado com o tipo *etc_t*, o novo arquivo herda o mesmo tipo. A Figura 3.1 demonstra o contexto de separação dos processos, arquivos, *roles* e domínios no sistema durante o *boot* a partir do *init*.



Fonte: [20]

Figura 3.1: Demonstração da separação de domínios a partir dos arquivos e processos envolvidos no *boot*, com seus respectivos *roles* no sistema. Nota-se que as fases do *boot* foram isoladas entre si. Um processo capaz de executar o servidor de `ssh` não será capaz de executar qualquer outro processo, pois este estaria fora do seu domínio. Nem mesmo o `init`, que é um processo com permissões elevadas, tem o direito de executar qualquer outro arquivo senão aqueles envolvidos no *boot*.

As políticas de segurança do *SELinux* são escritas a partir do nível de usuário e compiladas para o nível do *kernel*, para assim estarem prontas para serem utilizadas. Essas políticas são aplicadas em dois regimes, *Enforcing* e *Permissive*. No modo *Enforcing*, o *SELinux* impede toda ação que infringe uma política de segurança e a reporta como um evento a ser logado; já no modo *Permissive*, os eventos são apenas reportados.

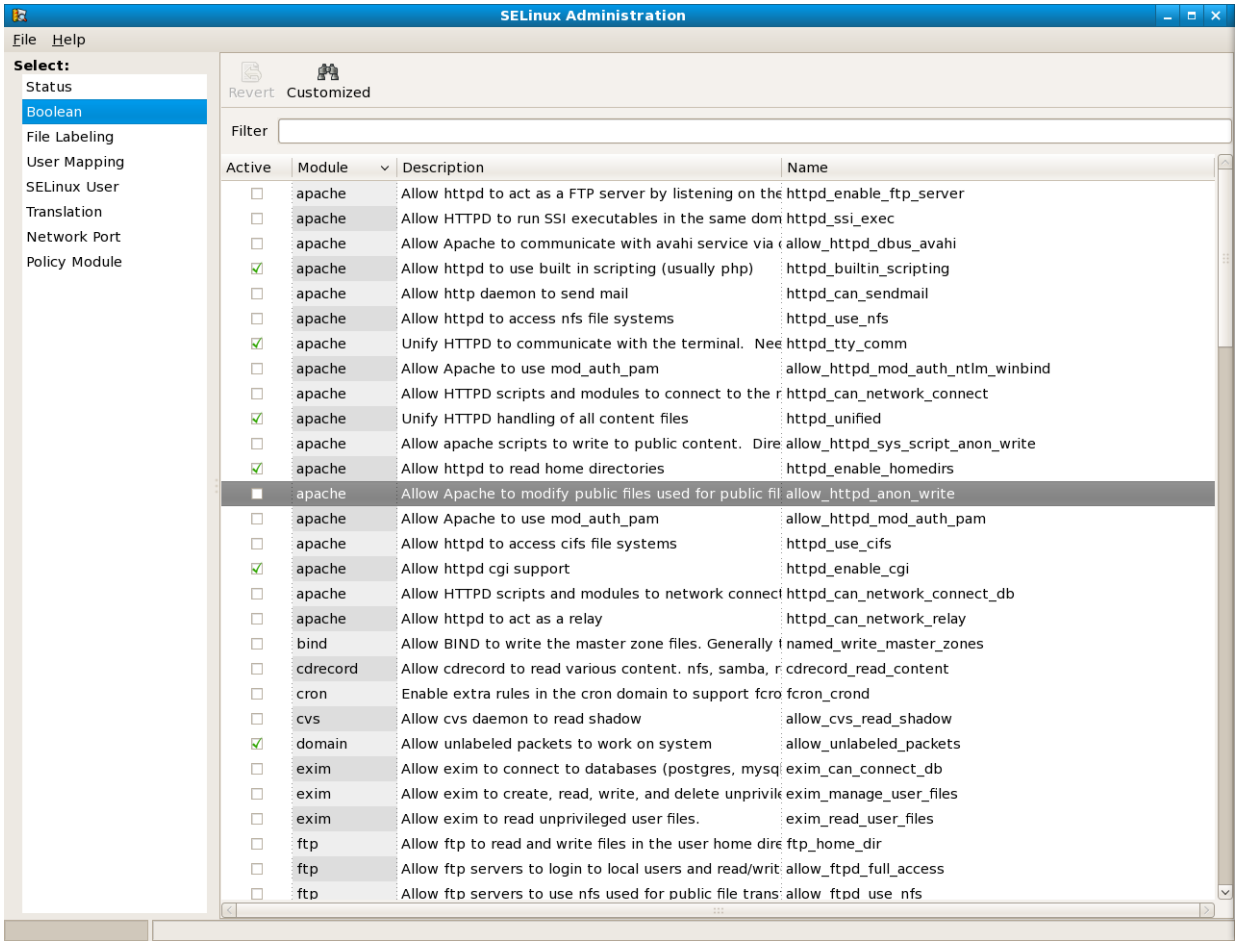
As políticas são um conjunto de regras que regem todo o *SELinux*, como quais *roles* que um usuário tem acesso, quais *roles* podem acessar quais domínios e que domínios podem acessar quais tipos de objetos. Essas políticas podem ser ajustadas ou criadas para adequar o comportamento de todos os processos do sistema, onde cada operação deva ser explicitamente permitida, em um arquivo de política, para não ser negada.

Apesar de complexas, as políticas do *SELinux* são flexíveis, permitindo regras brandas ou estritas para os processos. Entretanto, o conjunto de regras e políticas do *SELinux* é considerado extremamente complexo quando comparado a outros sistemas que fazem uso do *LSM* e cumprem o mesmo objetivo. Mesmo que o pacote do *SELinux* inclua ferramentas automatizadas para construção de políticas, o resultado final é sempre composto de centenas de linhas. A Figura 3.2 demonstra uma dessas ferramentas para gerenciamento de regras no *SELinux*.

A tomada de decisões do *SELinux* começa quando um processo tenta acessar um objeto, *e.g.*, um arquivo, para então o *SELinux policy enforcement server* verificar o *Access Vector Cache (AVC)*, onde as permissões sobre objetos são armazenadas em cache. Se uma decisão não pode ser feita com base nos dados do *AVC*, o *enforcement server* continua a busca a partir do contexto de segurança do executável do processo e o arquivo a ser acessado. Essa busca é feita em uma matriz de permissões, construída a partir das políticas de seguranças compiladas para aquele executável, identificado pelo seu *inode*. Caso o contexto do acesso faça *matching* em uma permissão registrada, o acesso é concedido, do contrário uma mensagem de acesso negado é logada.

3.3.2 AppArmor

O *AppArmor*, que utiliza um modelo *MAC* e também faz uso do *LSM*, foi projetado para ser um *security framework* o mais simples de administrar quanto possível, possuindo regras simples, flexíveis e de fácil entendimento. Diferentemente do *SELinux*, o *AppArmor* utiliza em suas políticas *(profiles)* as abstrações comuns ao estilo *Unix*, como *pathnames*, expressões regulares e nomes de usuário. Esses *profiles* são válidos para quaisquer processos cujo executável esteja em um caminho ou *path* específico no sistema de arquivos. É nesses *profiles* que estão contidas todas as políticas de acesso, objetos controlados e mecanismo de acesso disponíveis para o executável em questão. A aplicação das políticas não depende dos usuários que mantêm os processos, pois todos eles se encontram no mesmo conjunto de regras quando estão executando o mesmo programa. Outra diferença com o *SELinux* é a relação dos sistemas de arquivos suportados. Como o controle do *Apparmor* é feito sobre um caminho no sistema de arquivos, ele não depende de qual sistema de arquivos está sendo utilizado, diferentemente do SELinux que precisa de um sistema de arquivos com suporte a *security labels* e que não pode proteger recursos em sistemas de arquivos distribuídos.



Fonte: [62]

Figura 3.2: Ferramenta utilizada para composição de regras e políticas para programas reconhecidos como instalados no sistema. A figura demonstra parte das políticas utilizadas para assegurar o funcionamento e segurança do *Apache*.

Como o *AppArmor* aceita *wildcards* nos *profiles*, é possível compor regras mais genéricas para, por exemplo, liberar acesso apenas a arquivos com extensão *.log*. Outro uso importante dessa flexibilização nos *profiles* é a construção de regras e políticas especificamente para arquivos de linguagens interpretadas ou *scripts*, como `Perl`, `PHP` ou `Bash`. Este *framework* também possui um modo de aprendizado, onde o comportamento de um aplicativo pode ser observado e convertido automaticamente em um perfil de segurança.

O *AppArmor* possui dois modos de operação presentes em um *profile*, que são o *complain* e o *enforce*. O primeiro apenas registra violações às regras e políticas de segurança, enquanto o outro gerencia o acesso aos objetos controlados de acordo com suas políticas. É possível utilizar o *log* resultante do modo *complain* para criar um novo *profile* baseado no comportamento do processo em questão e assim, tanto resolver conflitos de segurança em um *profile* predefinido quanto criar um novo para uma aplicação específica.

A seguir tem-se um exemplo de um *profile* para o Apache, com suporte a *VirtualHost* e módulos, permitindo livre uso de bibliotecas compartilhadas.

```
#include <tunables/global>
```

```
/usr/sbin/apache2 {
#include <abstractions/apache2-common>
#include <abstractions/base>
#include <abstractions/nis>

capability dac_override,
capability dac_read_search,
capability net_bind_service,

/var/www/** r,
/data/www/safe/* rw,
deny /data/www/unsafe/* r,
/etc/apache2/apache2.conf r,
/etc/apache2/conf.d/charset r,
/etc/apache2/httpd.conf r,
/etc/apache2/mods-available/** r,
/etc/apache2/mods-enabled/** r,
/etc/apache2/ports.conf r,
/etc/ssl/openssl.cnf r,
/var/log/apache2/access.log w,
/var/log/apache2/error.log w,
/usr/lib/** mr,
/usr/sbin/apache2 mr,
/usr/share/apache2/** r,
/usr/share/file/* r,
/var/run/apache2.pid rw,

}
```

Apesar de relativamente simples, esse *profile* fornece as permissões necessárias para que o *Apache* execute sem conflitos de permissões, devido às restrições impostas. Mesmo que os *profiles* do *AppArmor* sejam significativamente menores que os arquivos de políticas do *SELinux*, ambos os sistema oferecem o mesmo grau de segurança. Os autores de [63] suportam essa afirmativa, caso ambos os sistemas sejam corretamente configurados e suas políticas tenham igual grau de permissividade. O trabalho citado vai um pouco mais além de comparar os *security frameworks*, pois nele propõem um esquema baseado em *FBAC (Functionality-Based Application Confinement)* [64] para composição das políticas, sem interação do administrador ou usuário do sistema.

Apesar das diferenças na configuração, aplicação dos *labels* ou identificação dos objetos e processos protegidos, os *frameworks* que usam o *LSM* são equivalentes em termos de segurança fornecida. Mesmo que eles não utilizem o mesmo modelo de gerência de acesso, como o *Tomoyo Linux* que suporta *RBAC*, o *LSM* fornece o mesmo nível de acesso e controle para todo e qualquer *security framework* que utilize sua interface. Essa equivalência é reciproca para as limitações do *LSM*, tais como:

- Proteção dependente das políticas; novas vulnerabilidades, novas políticas.

- Restrições aplicadas apenas a objetos críticos; *file system*, IPC(*sockets, queues etc.*), *kernel capabilities*.
- Não essencialmente privacidade dos dados do processo; pai-filho ou processos de um mesmo executável, mesmas permissões.

3.4 Grsecurity

O *Grsecurity* [73] é um sistema de segurança fortemente baseado nos princípios do *OpenBSD*, em que a abordagem difere da usada por sistemas baseados no *LSM*, pois faz seus próprios *hooks* e pontos de entrada no *kernel*. Apesar de também se basear no modelo *MAC*, sua proteção vai além: como o *Grsecurity* é constituído por um conjunto de *patches* do *kernel Linux*, ele faz mudanças de segurança no código do *kernel* em quase todos os subsistemas, aumentando a proteção nos níveis de usuário e de *kernel*.

Com essa abordagem, o *Grsecurity* protege todo o sistema contra *exploits*, ataques inerentes ao funcionamento do *kernel Linux* (*e.g.*, *jitspray* [37]) e mesmo a manipulação dos dados de um processo por outro. Essa manipulação inclui a visão de quaisquer entradas no *procfs*, lista de processos em execução e uso de *features* de auditoria e *debug*, como o `gdb`, `ptrace` ou `mmap`. Ele também permite que o administrador, entre outras coisas, defina uma política de privilégios mínimos para o sistema, em que cada processo e usuário tenha apenas os menores privilégios necessários para o ambiente.

Um dos principais componentes incluídos no pacote do *Grsecurity* é o *PaX* [82], que não é desenvolvido pelo mesmo grupo que mantém o *Grsecurity*. O *PaX* é empacotado como parte do *security framework* para fornecer mecanismos de proteção para o nível de usuário contra a maioria dos ataques de execução de código não pretendido. Esses ataques são aqueles que garantem leitura ou escrita no espaço de endereçamento de um processo atacado, *e.g.*, *stack* ou *heap overflow*, *format string*, etc. Para evitar tais ataques, o *PaX* introduz no *Linux* uma série de conceitos já mencionados neste trabalho, tais como o *eXecute Disable* nas páginas de memória e *ASLR*, além de outros como *gcc plugins galore* [81].

O subsistema do *Grsecurity* responsável pela gerência de segurança de processos em nível de usuário possui dois modos de operação: aprendizado e produção. No modo aprendizado, o sistema é mantido fora de um ambiente "não confiável" (produção), no qual todas as operações realizadas pelos processos são ditas confiáveis e farão parte da base de conhecimento, utilizada na composição das regras e políticas de acesso. Para determinar o comportamento esperado de um processo e a decisão a ser tomada em caso de anomalia, o *Grsecurity* não utiliza uma configuração com regras específicas por executável, mas sim um aprendizado de máquina com regras baseadas no *RBAC* (*Role-Based Access Control*).

O *RBAC* é uma abordagem de gerência de permissões ao sistema, através do controle de acesso a arquivos, *capabilities*, *sockets etc.*, a todos os usuários, incluindo o *root*. Esse sistema é uma implementação no *Grsecurity* que associa papéis (*roles*) para cada usuário e processos. Cada *role* define quais operações podem ser realizadas em determinados objetos, de acordo com a regras e políticas aprendidas pelo *security framework* ou definidas

pelo administrador. Dada uma coleção de *roles* e operações, os processos serão restritos a executarem somente as ações que não fogem do *role* de seus usuários, mesmo que estes tenham permissão pelo *DAC*. Além disso, o *Grsecurity* utiliza a política de negar por padrão tudo aquilo que não foi declarado ou aprendido pelo sistema, garantindo assim que um usuário não execute ações nunca antes vistas pelo sistema ou prevista pelo administrador.

Essa gerência de permissões força um atacante a reavaliar seus métodos de ataque, uma vez que o acesso à conta *root* não significa que ele terá acesso completo ao sistema. O acesso a um objeto em particular pode ser explicitamente concedido aos processos que dele necessitam, de tal modo que *root* atua como qualquer outro usuário. Embora o *Grsecurity* e seu sistema *RBAC* não forneçam uma segurança perfeita, eles aumentam consideravelmente a dificuldade de se comprometer com sucesso o sistema.

Além dos mecanismos de proteção já listados, o *Grsecurity* é capaz de fornecer um conjunto extenso de *features* de segurança ao sistema. A lista completa dessa proteção e sua devida descrição encontra-se em [72]. Os principais tipos de proteção alcançados por esse sistema são:

- Cheques de limites nas cópias de/para entre *kernel* e nível de usuário.
- Impede o acesso direto entre nível de usuário e o *kernel*.
- Impede estouros na pilha do *kernel*, em arquiteturas de 64 *bits*.
- *Hardening* nas permissões de memória do nível de usuário.
- *Padding* aleatório entre as pilhas das *kernel threads*.
- Resposta automática contra *kernel exploit bruteforcing*.
- *Chroot hardening*.
- Elimina ataques de *side-channel* contra os terminais de administradores.
- Evita uso do `ptrace` para bisbilhotar outros processos.
- Evita o auto-carregamento de módulos vulneráveis no *kernel*.
- Evita o *dumping* de binários não passíveis a leitura.
- Assegura privilégios consistente em ambiente *multithread*.
- Nega acesso aos objetos de *IPC* demasiadamente permissivos.
- Randomização do *layout* das estruturas de *kernel*.
- Protege os argumentos de funções de sofrerem *integer overflow*.
- Adiciona entropia do *kernel* nos primeiros estágios de *boot*.
- Torna estruturas sensíveis do *kernel* como somente leitura.
- Garante que todas os apontadores do *kernel* apontem para o código do *kernel*.
- Evita o vazamento de dados da *stack* entre instâncias de *syscalls*.

A matriz de comparação presente em [71] torna possível visualizar as diferenças entre *Grsecurity*, *SELinux* e *AppArmor*. Por exemplo, *Grsecurity* pode proteger um processo tal como qualquer sistema baseado em *LSM*, mas como protege o *kernel*, acaba por proteger também o subsistema do próprio *LSM*. Já [14] compara a qualidade da segurança provida entre *Grsecurity* e *SELinux*.

3.5 Mecanismos de segurança acadêmicos

Além dos mecanismos e tecnologias utilizados pela indústria, foram pesquisados sistemas de segurança propostos pela academia, capazes de resolver instâncias do problema de segurança e privacidade de um processo. O mecanismo proposto e desenvolvido nesta dissertação foi também influenciado pelos sucessos dos sistemas de segurança conceituais, assim como a resolução de suas limitações. Também foram estudadas abordagens feitas por outros sistemas operacionais, como o *Haiku* [23] e *QubesOS* [83]. *QubesOS* é uma distribuição orientada para a segurança baseada no *microkernel* do *Xen*, que utiliza o *user-level* Linux. Seu principal conceito é segurança pelo isolamento usando domínios *Xen*, implementados como máquinas virtuais leves para isolar os vários subsistemas entre si. Esse sistema operacional permite que grupos de processos, isolados em domínios diferentes, estejam separados em nível de *hardware* lógico, devido à virtualização dos domínios.

Recentemente, diversas pesquisas sobre segurança em sistema operacionais vêm migrando do universo *mobile* para o uso em *desktop*, como conceitos de segurança e permissões de usuários. Os autores de [36] propõem uma normativa de segurança nas linguagens de montagem, capaz de fornecer ao programador uma segmentação de permissões e comportamentos que um *software* deva ter. Desta forma, assim como no universo *mobile*, um *software* no *desktop* terá suas limitações definidas em tempo de compilação, bastando ao sistema operacional supervisioná-lo. Outro trabalho que utilizou da segurança em sistemas operacionais *mobile* foi [46]. Os autores desse artigo propõem um *Input-Driven Access Control*, onde um sistema intermedia o acesso das aplicações de um usuário aos recursos sensíveis do sistema operacional, como o microfone, câmera, *clipboard* e as janelas de outras aplicações. Essa intermediação ocorre com uma notificação *pop-up* ao usuário, para questionar o acesso de uma aplicação a um recurso considerado crítico.

Dos trabalhos relacionados, dois destacaram-se pelo uso em *Linux* por tratarem da privacidade dos dados de um processo e serem considerados sólidos para resolverem alguns ataques aqui demonstrados. Esses trabalhos foram o *PrivExec* e o *CryptKeeper*.

3.5.1 PrivExec

O *PrivExec* [45] se utiliza de múltiplos mecanismos para garantir o isolamento de um grupo de processos e suas dependências. Esses mecanismos são a criptografia dos arquivos manipulados e das páginas de *swap* e restrições na comunicação por *Inter Process Communication (IPC)*. O objetivo desse sistema é fornecer a execução privada de um processo como um serviço do sistema operacional, criando uma distinção lógica entre os processos públicos e privados. Essa distinção é feita internamente no *kernel*, com o uso de uma *flag* adicional na *task_ struct*.

Enquanto os processos públicos são executados normalmente quanto aos recursos compartilhados do sistema, os processos privados estão sujeitos a restrições de segurança especiais. O *PrivExec* permite a inclusão de processos em grupos privados, para que seja possível relaxar as politicas dentre os componentes do grupo. Esse mecanismo é principalmente utilizado para resolver dependências, em vista das restrições ao uso de *IPC* que

corrompem a execução de alguns processos.

Dentre os processos de um mesmo grupo, não existe um identificador explícito para separá-los. Um processo é dito pertencente a um grupo caso ele possua a mesma chave secreta em comum com outros processos do grupo. Essa chave, utilizada nos serviços de criptografia, é compartilhada pelo primeiro processo a integrar o grupo, o qual é o processo privado que necessitou da inclusão de outros processos para resolver suas dependências.

O isolamento do armazenamento persistente no *PrivExec* ocorre pela cifragem dos arquivos abertos para escrita pelo processo privado e pela cifragem das páginas de *swap*. Essa cifragem utiliza uma versão alterada do *eCryptfs* [33], onde cada arquivo aberto é visto como um *security container*. Na implementação padrão do *eCryptfs*, cada contêiner é uma instância de um sistema de arquivos (lista de *inodes*, metadados e *super inode*). Para juntar os diversos contêineres privados em uma única árvore de diretórios, o *PrivExec* utilizou o *OverlayFS* [4]. O uso desse *union file system* permite a união de um *secure container* e sua árvore de diretórios com a raiz de um sistema de arquivos de armazenamento persistente, como o *ext4*.

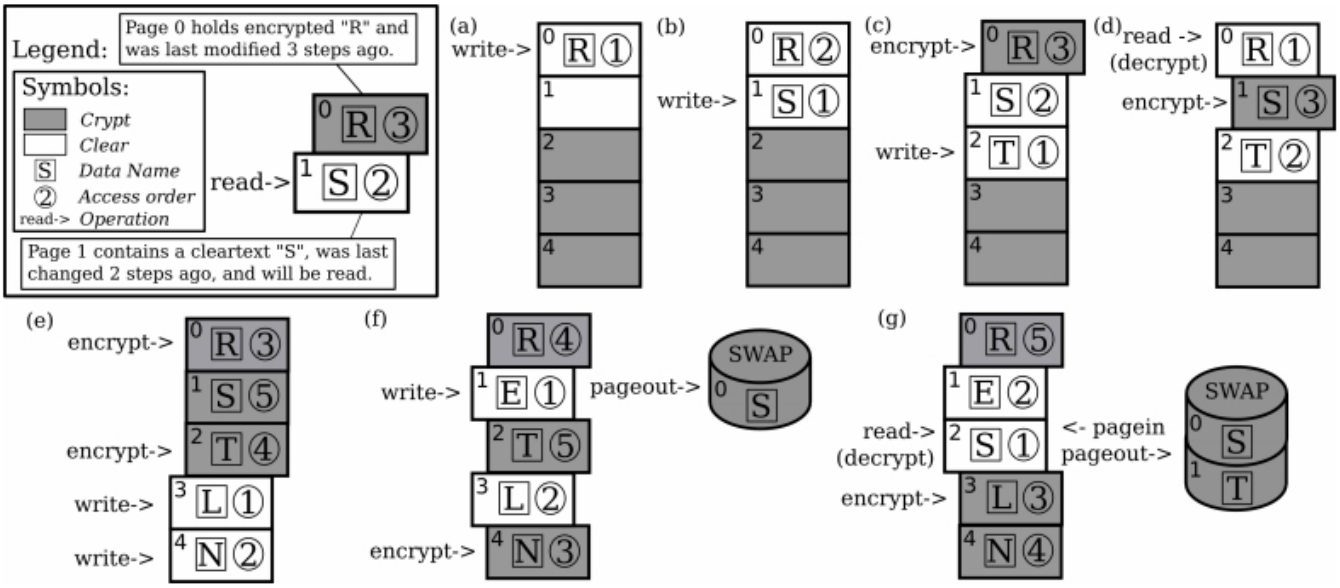
O *PrivExec* impõe restrições em relação ao uso de *IPC* para evitar vazamentos de dados do processo privado para fora de seu grupo. O identificador desse grupo é utilizado para controlar o ponto final da comunicação, antes da análise de permissões feita pelo *kernel*. Essa política permite que processos do mesmo grupo se comuniquem livremente, enquanto outros processos recebem um erro de permissão quando tentam iniciar uma comunicação com outros grupos privados. As estruturas do *Linux* que foram modificadas para respeitar essa política foram: *UNIX SysV & POSIX shared memory, message queues* e *UNIX domain sockets*.

O *PrivExec* consegue o isolamento de um processo arbitrário, garantindo o sigilo dos arquivos manipulados, tanto no disco quanto no *swap*, e da comunicação via *IPC*. Entretanto, essa implementação apenas provê segurança nos protocolos de *IPC* mais comuns, assim como não garante qualquer isolamento contra leitura ou vazamento de memória perante outros processos. As limitações do *PrivExec* incluem o fato de ser uma proposta conceitual na qual não se considera resistência em caso de ataques, falhas do sistema ou presença de *malware* previamente à sua instalação.

3.5.2 CryptKeeper

O sistema *CryptKeeper* [51] divide a memória principal em dois segmentos: *Clear*, menor, porém de tamanho ajustável, rápido e inseguro; *Crypt*, maior, lento e cifrado. Esse sistema é implementado como uma variação do gerenciador de memória do *kernel Linux*, onde sua função é cifrar páginas do *Clear*, tornando-as parte do *Crypt* e posteriormente decifrar páginas do *Crypt* para o *Clear*. Assim, *CryptKeeper* fornece cifragem de memória de maneira transparente para todos os processos em execução. O segmento *Clear* contém as regiões de memória recentemente utilizadas, seguindo uma política de gerência similar à de um cache. Tendo o seu tamanho configurável, o *Clear* é utilizado como um cache ao *Crypt*, oferecendo uma troca entre segurança e desempenho, tendo em vista que quanto maior a primeira, menor o esforço do módulo de criptografia utilizado pelo *kernel* e menor a segurança pelo número reduzido de áreas de memória cifradas.

O *CryptKeeper* utiliza uma chave secreta gerada na criação de cada processo, para assim cifrar as páginas no *Clear* e decifrar aquelas do *Crypt*. O tamanho da seção *Clear* é definido em número máximo de páginas, de acordo com a configuração do sistema de segurança. Desta forma, assim que o número total de páginas de um processo ultrapassar aquele máximo, a ultima página utilizada será cifrada e integrada ao *Crypt*. A decifragem ocorre apenas quando o processo requisita uma página que esteja no *Crypt*. Quando isso ocorre, o *CryptKeeper* coloca uma página, pela política *LRU*, no *Crypt* e a página requisita no *Clear*. A Figura 3.3 ilustra o processo de entrada e saída de páginas das seções do *CryptKeeper*:



Fonte: [51]

Figura 3.3: Exemplo de operações do *CryptKeeper*. Com tamanho máximo do *Clear* em duas páginas, as operações em *a* e *b* respeitam esse tamanho. Em *c* a inserção de *T* leva a cifragem de *R* pela política *LRU*. A página *R* volta ao *Clear* quando acessada em *d*. Já em *e* a escrita das duas novas páginas transfere as outras para o *Crypt*. Com a inserção de *E* em *f*, a página *S* é transferida para o *swap*, por falta de páginas de memória na *RAM* e *N* para o *Crypt* por ser a terceira página em *Clear*. Por fim, em *g* a página *S* é acessada, levando a transferência de *T* para o *swap* e *L* para o *Crypt*.

O *CryptKeeper* destina-se a proteger os processos contra vazamentos de dados fora de seus contextos, como os feitos através do *Cold Boot Attack* [19] ou falhas que permitam abusar da leitura de áreas de memória arbitrárias pelo *kernel*. Assim, quando o vazamento é feito de dentro do contexto do processo, o *CryptKeeper* não oferece nenhuma proteção. Alguns desses vazamentos são a leitura da memória do processo através de *IPC*, a injeção de uma biblioteca dinâmica no contexto do processo ou os ataques listados no Capítulo 2. Além disso, de acordo com os autores desse sistema, o *overhead* na frequente troca de páginas torna o sistema aplicável apenas para ambientes *desktop*.

3.6 Limitações dos trabalhos estudados

Os *security frameworks* apresentados representam o estado da arte na proteção de processos, seja contra ataques por meio de falhas de *software* ou contra ataques que mudam o comportamento esperado do programa ao longo de sua execução. Embora tais *frameworks* possam ser bem-sucedidos na prevenção da exploração de muitas vulnerabilidades, existem classes inteiras delas que não podem ser evitadas, como *kernel capabilities* desnecessárias a um processo ou *race conditions*.

Isto ocorre devido parcialmente à limitação inata de todos os sistemas de controle de acesso, como descrito por [2]: eles precisam ser tão automáticos quanto possível e possuir configurações legíveis e facilmente editáveis, além de um rigoroso e estrito controle de objetos, de modo a fazer cumprir as políticas de segurança ao eliminar erros do administrador. Desta forma, ocasionalmente uma vulnerabilidade precisa ser tratada pela elaboração de novas regras, em geral porque novos objetos ou *capabilities* tornaram-se alvo dessa vulnerabilidade e as antigas regras não foram suficientes. Essa limitação está presente no *Grsecurity*, conforme mostrado por [5] ao fazerem uma profunda análise, incluindo limitações e *trade-off*, da gerência por *RBAC* em relação a *MAC* do *LSM*. Em [15], corrobora-se tais limitações ao se contrastar a implementação do *SELinux* e do *Grsecurity* quanto à gerência de acesso.

Uma gerência de acesso que não partilha das falhas estruturais, quanto ao reforço da política por permissões presentes em sistemas *DAC*, *MAC* ou *RBAC*, é o modelo *Attribute-Based Access Control (ABAC)* [21]. Uma abordagem dessa gerência foi utilizada no sistema de segurança desenvolvido, por ter uma superfície de ataque menor, como demonstrado em [89].

Mesmo que seja utilizado um sistema cuja gerência de acesso não seja um *MAC*, como os trabalhos [64] ou [46] propõem, não há garantias que esse sistema de controle resolva aquelas limitações. Em vista do código de tomada de decisões e políticas residir no lado do *kernel*, o comprometimento deste pode facilmente resultar no comprometimento do sistema de controle de acesso. Como diversos ataques exploram características do *kernel*, é possível que nenhum gerente de acesso consiga impedir o ataque em andamento, antes que o nível de *kernel* seja comprometido. Os desenvolvedores do *Grsecurity* deixam claro em [80] que, mesmo com seu sistema contendo vários recursos que ajudam a prevenir explorações do *kernel* e o tornam um ambiente hostil para o atacante, ele não é imune a essa limitação fundamental.

Ataques ao *kernel* ou ao gerente de acesso em nível de usuário que perpassam os *security frameworks* existem e não são raros. O *SELinux* é alvo frequente de um ataque [34] inerente à sua arquitetura em nível de usuário, capaz de desabilitar o modo *Enforce* e logo toda a aplicação de políticas no sistema. Tal ataque é atualizado com frequência (sua última atualização registrada aqui tendo ocorrido em 21 de Junho de 2016) para corresponder às ultimas atualizações e novas políticas do *SELinux*, utilizadas justamente para barrá-lo. Uma abordagem [66] desse ataque é utilizada para desabilitar o modo *Enforce* do *SELinux* no *Android* ≥ 5.

Apesar de não existir uma falha contra o *AppArmor* desde 2009, o último ataque [85] conhecido capaz de passar pelo controle do *AppArmor* permitiu elevação de privilégios.

Essa elevação de privilégios utilizou uma gerência errada de permissões no *Overlayfs*, para criação de *user namespace* e pontos de montagem virtuais. Também era possível acessar arquivos sem a elevação de privilégios, como */etc/shadow*, mesmo se houvesse uma política do *AppArmor* impedindo essa ação. Isso ocorreu devido ao acesso no *user namespace* ter um *path* diferente do arquivo real, *e.g.*, */etc/shadow* ser acessado como */media/mount_ user1/file.txt*. Um dos autores do *Grsecurity* mantém um repositório [70] (desatualizado desde 2014) de *exploits* feitos por ele contra os *security frameworks SELinux* e *AppArmor*. Esse repositório foi uma maneira de chamar a atenção para as falhas existentes naqueles sistemas e para o próprio *Grsecurity*.

Apesar de ser um gerente em nível de *kernel* como os demais, o *Grsecurity* nunca teve uma falha crítica publicamente conhecida. As abordagens capazes de explorar, de alguma forma, as estruturas desse sistema de segurança, sempre se basearam na premissa da existência de uma falha que permitisse a escrita arbitrária na memória do *kernel*. A última abordagem conhecida [44] necessitou da existência de uma falha para escrita arbitrária na memória do *kernel* e de uma falha para *kernel stack memory disclosure*. Apesar dela possibilitar a manipulação de algumas *flags* e estruturas do *Grsecurity*, um atacante com os poderes exigidos para esse ataque conseguiria subverter o sistema de outras maneiras, como descrito na resposta [69] ao ataque por *Spender*, principal mantenedor desse sistema de segurança.

Além das limitações descritas, cabe ressaltar que os *frameworks* acima mencionados não tratam dos recursos dos processos controlados por eles em relação a outros processos do sistema, caso esses últimos também não sejam controlados. Ambos, *LSM* e *Grsecurity*, controlam o que um processo controlado pode acessar, porém sem interferir no que um processo com os mesmos privilégios do processo controlado pode acessar, caso esse acesso não viole nenhuma política de segurança. Esse modo de projeto é um problema mais de privacidade do que de segurança: caso um processo controlado seja subvertido, ele não será capaz de fazer mais do que lhe é permitido pelas políticas de segurança. Porém, é possível um outro processo obter dados daquele processo controlado sem de fato influenciar em sua execução, isto é, sem causar correspondência com nenhuma regra de segurança, dado que não se trata de um cenário identificado como ataque pelos *frameworks*.

Um bom exemplo é o servidor Web *Apache*, o qual possui um único arquivo de configuração de políticas nos *frameworks* supracitados, mas geralmente é executado como mais de uma instância. Cada instância cuida geralmente de vários *virtual hosts* e é um processo de um mesmo usuário. Neste cenário, uma instância pode interferir com dados de outras instâncias, pois para os sistemas de segurança, todas as instâncias têm o mesmo nível de acesso e privilégios. Isso acontece devido à forma como cada processo é controlado com base no executável que lhe deu origem, através do *path* ou *inode* no sistema de arquivos. Logo, uma instância do *Apache* pode sobrescrever arquivos de outra instância, tal como *logs*, *Unix socket lockfiles*, configurações ou mesmo os arquivos hospedados.

Embora o *Grsecurity*, diferentemente do *LSM*, não possua apenas um gerente de acesso para aplicar suas políticas e possa tratar cada processo como uma instância diferente no sistema operacional, ele não irá negar o acesso de uma instância à outra. Isso acontece porque o *Grsecurity* não pode coincidir esse comportamento com um perfil de ataque, fazendo com que apenas verifique os direitos de acesso dos recursos que o processo está

autorizado a utilizar. Assim, ele também não consegue garantir a privacidade dos dados de um processo quando o acesso a estes é inerentemente permitido para outro processo. Este cenário sempre acontece quando o projeto do sistema operacional espera uma base de confiança na relação entre processos, tais como a relação pai-filho ou processos do mesmo usuário.

3.6.1 Sumário dos Trabalhos Relacionados

A Tabela 3.2 apresenta os mecanismos de segurança estudados e apresentados neste Capítulo, de acordo com as respectivas características de segurança identificadas no presente trabalho. Essas características podem ser integralmente ou parcialmente cumpridas por esses mecanismos, assim como podem não existirem por completo. O mecanismos proposto no presente trabalho, o OSPI (*Operating system Security Process Isolator*), foi incluído ao lado dos demais mecanismos afim de mapear suas características e limitações, além das falhas sanadas por ele quanto aos outros mecanismos de segurança.

Os campos na Tabela 3.2 com o símbolo "✓" indicam que o mecanismo fornece completamente a proteção ou isolamento em questão. Aqueles com o símbolo "●" provem algum nível de segurança, mas não completamente (*e.g.*, proteção em alguns dos protocolos de *IPC*). Já os mecanismos marcados com "✗", não provêm qualquer restrição ou proteção na característica em questão.

As características foram divididas em proteções contra ataques, isolamento das estruturas do sistema operacional ou do *hardware*, presença de um modelo de controle de acesso (*e.g.*, *MAC*, *RBAC* etc.) ou a resiliência do mecanismo contra ataques ao próprio código. Todas as características referentes ao OSPI serão detalhadas nos Capítulos 4 e 5.

Característica	Grsecurity	LSM	OSPI	QubesOS	PrivExec	CryptKeeper
Access Control	√	√	√	χ	χ	χ
Prot. *userland* mem. *leak*	√	•	√	√	χ	√
Prot. *kerneland* mem. *leak*	√	χ	•	√	χ	•
Prot. *userland exploits*	√	•	√	√	χ	χ
Prot. *kerneland exploits*	√	χ	χ	√	χ	χ
Prot. abuso permissões	√	√	√	√	√	χ
Cripto. memória	χ	χ	√	χ	χ	√
Cripto. arquivos	χ	χ	√	χ	√	χ
Isola *IPC*	√	χ	•	√	•	χ
Isola nível de *kernel*	•	χ	χ	√	χ	χ
Isola nível em de *HW*	χ	χ	χ	√	χ	χ
Isola *tasks* mesmo usuário	•	•	√	•	χ	χ
Isola *tasks* mesmo executável	χ	χ	√	√	χ	χ
Isola *user-thread*	√	χ	•	√	χ	χ
Isola *kernel-thread*	√	χ	χ	√	χ	χ
Resiliência à ataques	√	•	•	√	χ	χ

Tabela 3.2: Tabela comparativa das características de segurança presentes em cada um dos mecanismos de segurança. Um mecanismo pode preencher completamente ou parcialmente a característica de segurança, assim como não cumpri-la de nenhuma forma.

As duas principais características do OSPI, que os outros mecanismos não cumpriram perfeitamente, foram os isolamentos de processos de um mesmo executável ou usuário. Isso deve-se a abordagem de segurança utilizada no mecanismo, que permitiu a mudança do comportamento da camada de *syscall* de acordo com o processos que a utiliza. Assim, tornando possível diferenciar *tasks* ou processos de um mesmo executável (ou usuário), de tal maneira a isola-los no acesso aos objetos pertinente a execução exclusiva de cada processo, mesmo quando as permissões desses objetos são compartilhadas pelo sistema operacional ou pelos outros mecanismos de segurança.

O mecanismo proposto neste trabalho sanou algumas das limitações presentes em outros mecanismos (principalmente *Grsecurity* e *LSM*), ao retirar a proteção por meio de gerências de acesso e por perfil de ataques, passando-a para a gerência da posse dos dados. Por exemplo, de acordo com a gerência de permissões de um objeto, como um arquivo ou um *socket*, este só pode ser acessado caso o processo consiga decifrá-lo e não de acordo com seus privilégios no sistema operacional.

Capítulo 4

Mecanismo de Proteção Proposto

Neste capítulo, propõe-se um mecanismo de isolamento de processos arbitrários a partir do contexto de outros processos e apresenta-se o protótipo desenvolvido. O objetivo de tal mecanismo é aumentar a segurança e a privacidade do usuário mesmo no caso de processos que compartilham os mesmos privilégios. A ideia por trás dessa proposta é a de complementar o tipo de proteção provida por outros *frameworks*, como o *Grsecurity*. Desta forma, o mecanismo de segurança proposto resolve algumas das limitações presentes em outros mecanismos sem criar a necessidade de toda uma infraestrutura de segurança nova, principalmente quando a base de confiança do sistema operacional não é suficiente. O mecanismo de segurança aqui proposto, chamado a partir de agora de OSPI (*Operating system Security Process Isolator*), foi desenvolvido na forma de um módulo do *kernel Linux* para arquitetura x86 com suporte à extensão *AES-NI* e um gestor de nível de usuário.

A arquitetura do OSPI permite que os processos utilizem características especiais de segurança introduzidas no *kernel* pelo módulo, o qual tem por objetivo segregar os processos em grupos, de modo a aplicar políticas especiais de segurança. Apesar do OSPI implementar gerência de acesso baseada em permissões, a heurística utilizada é uma abordagem mais próxima do modelo *ABAC* do que do *MAC*. Isso ocorre devido à abordagem da gerência de acesso não ser baseada nas permissões do processo ou do usuário, mas sim na posse de um objeto. Cada objeto manipulado por um processo, em nível de usuário ou não, é exclusivamente de sua propriedade ou compartilhado entre um grupo predefinido de processos, caso este seja o primeiro processo a reivindicar a posse junto ao OSPI. Os mecanismos dessa posse devem primeiro respeitar uma máscara de permissões do OSPI, presente para cada processo controlado, antes mesmo de serem avaliados. Como a abordagem do modelo *ABAC* no OSPI (na gerência da posse do objeto) é aplicada antes das regras presentes na máscara de permissões do processo, o OSPI utiliza o modelo *MAC*, para gerenciar o mecanismo de acesso, somente após a heurística de posse do *ABAC*. A Figura 4.1 ilustra uma visão geral do OSPI, com todos os seus subsistemas, e as interações com o *kernel Linux*.

O OSPI faz uso de diversos recursos da *kernel internal API* para implementar *syscall hooking*, tais como *kthreads*, `k_queues`, `control groups`, *CryptoAPI* e de alguns símbolos do *kernel* como `cpu_debugreg`. Além das estruturas de *kernel* citadas, o OSPI também inclui as seguintes estruturas lógicas:

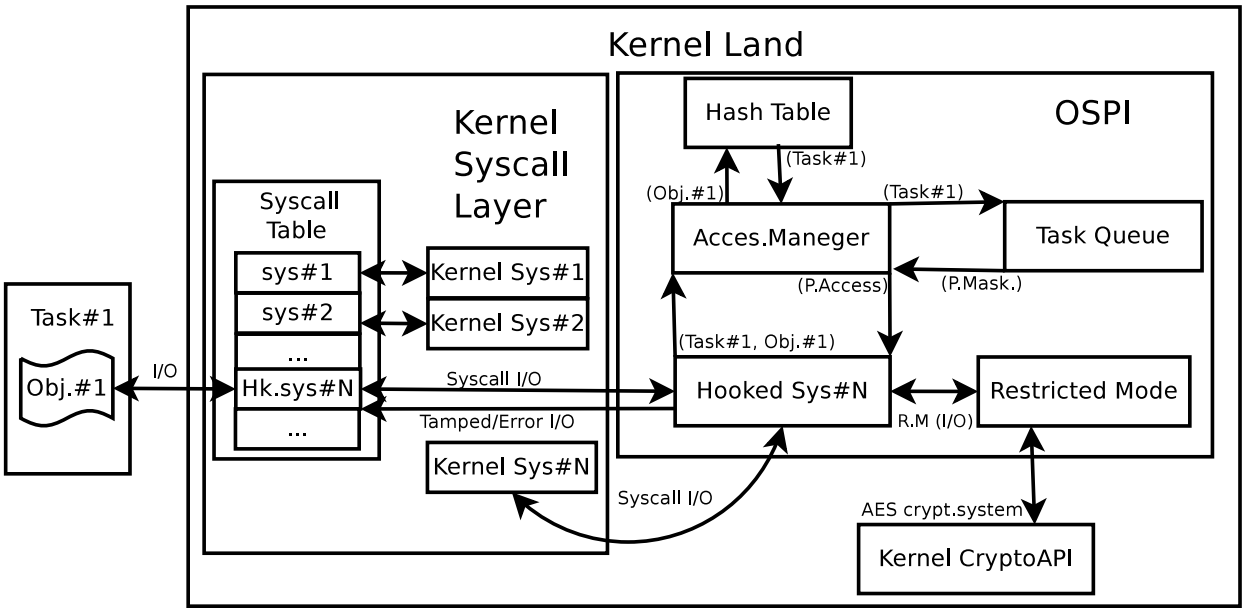


Figura 4.1: Visão geral do OSPI e seus subsistemas. O *hook* ocorreu na posição N da tabela de *syscall*, inserindo-se o endereço da função *Hooked Sys N*. É nessa função que todos os mecanismos de privacidade e isolamento se aplicam, antes e depois da chamada real da *syscall N*.

- Lista completa de processos controlados;
- Lista global de objetos controlados;
- Lista de símbolos do *kernel* por processo;
- Máscara por processo que mapeia as permissões e políticas de cada processo.

A lista global de objetos controlados é feita como uma tabela *hash*, cuja função é a indexação dos processos que detêm a posse de algum objeto. Essa tabela utiliza como chave o endereço virtual de memória onde encontra-se um objeto controlado e a função de espalhamento utiliza a divisão desse endereço. Esta divisão é feita em duas partes de 16 bits e uma com o restante dos bits para completar aquele endereço em que reside o objeto controlado. A Figura 4.2 mostra a organização dessa tabela, a qual torna o custo de busca assintóticamente superior no pior caso em $O(n)$, sendo n o número de objetos controlados, ou mantém no caso médio um custo constante. Assim, um dado acesso a um objeto, controlado ou não, terá pequena influência da heurística de busca sob o controle daquele objeto, minimizando o *overhead* do OSPI no sistema.

A lista completa de processos controlados é composta de nós duplamente encadeados apontados pela tabela *hash* descrita anteriormente. Cada nó possui uma lista de apontadores para, respectivamente: lista de *PIDs*, os quais identificam os processos (ou *tasks*) contidos no grupo daquele processo que detém a posse do objeto; lista de *kernel symbols*, que funciona como *black/white list* dos símbolos de *kernel* (*e.g.*, `cpu_debugreg`) permitindo o controle das *features* do *kernel* acessíveis ao processo através de seus símbolos; máscara de permissões, que contém todas as políticas de permissões aos mecanismos de acesso do OSPI. A Figura 4.3 ilustra essa lista de processos controlados.

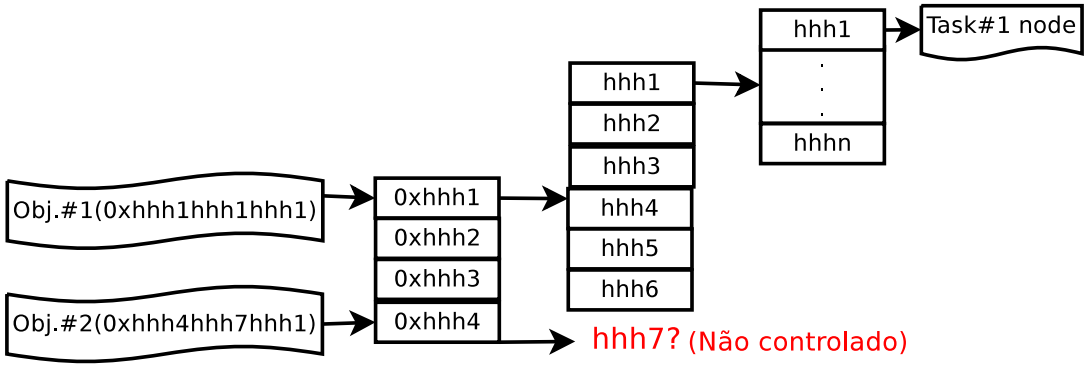


Figura 4.2: Tabela *hash* dos endereços de memória correspondentes aos objetos controlados, os quais indexam os processos que detêm a posse desses objetos.

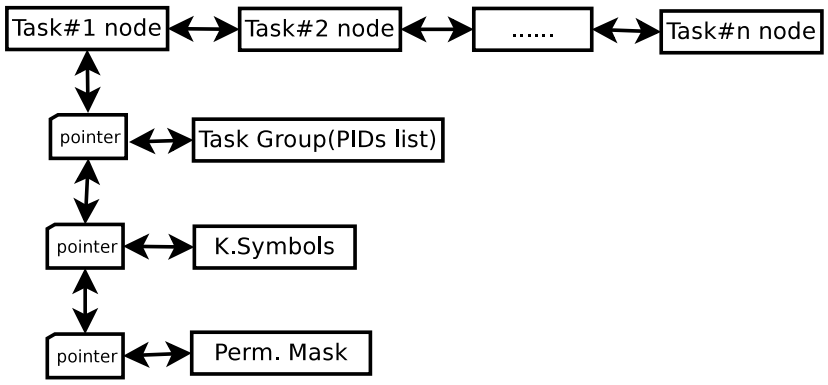


Figura 4.3: Lista de processos controlados pelo OSPI e a estrutura de nós dos processos controlados.

A utilização dessas estruturas, do *kernel* e do OSPI, permite que o módulo controle o acesso e as permissões de cada processo, ou grupo de processos, cujas permissões são compartilhadas. Esse controle é realizado nas estruturas cujo comportamento padrão deva ser alterado seguindo políticas de segurança, como a *syscall* `process_vm_readv`, que deve retornar sucesso, porém não deve retornar um bloco de memória quando destinada a um processo protegido, independentemente das permissões do usuário. As técnicas para interceptar e alterar as estruturas do *kernel* incluem o *hooking* ou o acesso direto aos subsistemas do *kernel*, como no *virtual filesystem*, impedindo o mapeamento de um *incore inode* por um outro módulo do *kernel*, através da retirada de sua referência na lista de *incore inodes*.

4.1 Arquitetura do OSPI

Ao contrário de outros mecanismos de segurança do Linux, o controle de recursos proposto não utiliza os locais dos objetos (*e.g.*, caminho no sistema de arquivos), mas sim as informações de posse presentes nas estruturas do módulo. Para obtê-las, o módulo decidirá se (e como) um recurso será compartilhado, de acordo com a máscara de acesso do primeiro processo que obtiver a posse do objeto. Essa posse é atribuída através da relação entre processo controlado e objeto a ser manipulado, dado que tal objeto não foi manipulado

por nenhum outro processo ou grupo de processos até então. A atribuição dessa posse ocorre na inserção do endereço de memória virtual do objeto na tabela *hash*, onde o campo indexado é o apontador para o processo controlado.

Para que um processo ou *task* passe a ser controlado, seu usuário deve inserir um vetor de *bytes* na interface do OSPI com o nível de usuário via *procfs* em `/proc/ospi`. Esse arquivo virtual consiste de uma entrada no *procfs*, onde o OSPI registra funções a serem chamadas nos eventos de *I/O*, quando feitos no (`/proc/ospi`). A referida interface permite que o nível de usuário se comunique com o OSPI em nível de *kernel* de maneira simples e ágil—há apenas uma troca de contexto aos eventos. A política de acesso a essa interface do OSPI obedece às seguintes regras:

1. Toda escrita deve conter 28 bytes; qualquer entrada de tamanho diferente é desalocada.

2. Apenas o usuário insere seus processos para proteção por meio da escrita do vetor de bytes.

3. A máscara de permissões pode ser incrementada, mas nunca decrementada; apenas o próprio processo pode tirar a si próprio do modo protegido, porém isso demanda conhecimento prévio do uso da interface pelo programador.

4. Somente o próprio processo consegue ler sua própria entrada no *procfs*; processos não protegidos recebem o valor *NULL* como dado de entrada na leitura.

5. Um processo em um grupo protegido recebe apenas a máscara de permissões do seu grupo durante a leitura; essa máscara foi inserida no vetor de bytes do processo protegido, o qual detém as posses desse grupo.

A Figura 4.4 ilustra a organização e o tamanho em bytes dos campos de entrada do vetor que representa a interface para o módulo de segurança OSPI, cujos campos são definidos da seguinte forma: o primeiro campo estabelece a operação a ser feita (inserção ou remoção); o segundo define qual processo, de acordo com seu *PID*, será incluído no esquema de proteção; o terceiro é o *PID* do processo que detém um grupo onde o processo a ser controlado será incluído; o quarto campo é a máscara de permissões, que inclui a *syscall* controlada, um apontador para uma lista de apontador e uma máscara de permissões do gerente de acesso.

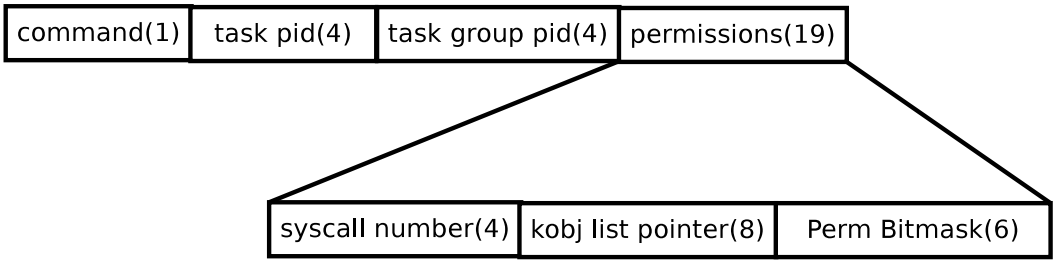


Figura 4.4: Organização do vetor de bytes utilizado como entrada da interface com o OSPI para definição do nó de um processo controlado.

A frequente mudança no projeto de subsistemas do *kernel Linux* associada a problemas na exportação de símbolos para módulos fomentou a decisão de se desenvolver um gerente de grupo próprio, em vez de usar o mecanismo nativo de grupos de processos (*Control Groups*) [48]. A abordagem utilizada na gerência de grupo é aquela onde um grupo é construído ao redor de um processo que detém a posse de um recurso, e o acesso ao recurso depende da máscara de permissão daquele processo. A política de inclusão de processos em grupos segue um regime estrito, devido à restrição quanto a confiança entre processos partilhada no sistema operacional. Um usuário consegue inserir seus processos no regime de proteção do OSPI, apenas durante uma janela temporal. Essa janela é calculada para ser aberta por 1000 *ticks* do escalonador de *tasks* ou por $\frac{1000}{CONFIG\_HZ}$ segundos, onde `CONFIG_HZ` é a frequência de *kernel ticks*. Além da política da janela de inclusão, processos adicionados a um grupo devem respeitar as seguintes regras:

1. Um grupo pode ser composto por processos de diferentes usuários, mas somente o usuário pode inserí-lo em um dado grupo.

2. Um processo pode somente pertencer a um único grupo.

3. Pode existir somente um processo por grupo com posses de objetos.

4. Caso seja requerida a proteção para um processo já contido em um grupo, sua inserção é negada até que ele seja removido do grupo.

As regras para remoção de processos sob proteção são mais simples: o usuário do processo protegido ou do processo contido no grupo pode remover o referido processo sem outros requisitos. Não existe reinserção, uma vez que fora da janela temporal um grupo não pode receber novas inserções.

A abordagem de controle é feita no acesso da camada de *syscall*, pelo fornecimento de uma interface. Esta concede o controle sobre cada processo no sistema, pois permite mudar como um conjunto de *syscalls* se comportam, de acordo com o processo que as executarem. Com isso em mente, se um processo tentar manipular qualquer objeto no espaço de usuário, seu pedido terá de passar primeiro pelo OSPI, pois os *hooks* são colocados em *syscalls* que fazem borda com a manipulação de arquivos, páginas de memórias e IPC, no nível de usuário. Devido ao conceito de protótipo do sistema, apenas um conjunto de *syscalls* foi incluído no controle do OSPI. Esse conjunto foi utilizado como estudo de caso, e prova de concepção do mecanismo, uma vez que o controle efetivo nessas *syscalls* representaria o sucesso do modelo proposto. As *syscalls* que levaram a implementação de suas funções de interposição foram as seguintes: `open`, `read`,`write`, `exec`, `mmap`, `msgget` e `msgsnd` . Com esses *hooks*, o módulo pode verificar se quem faz a chamada é um processo controlado ou se ele está manipulando recursos controlados, aplicando restrições quando necessário e, em seguida, passar a execução para o código original da *syscall*. Deste ponto em diante, outras estruturas de segurança, como o *Grsecurity*, podem aplicar as suas políticas definidas de modo que o OSPI irá fortalecer um mecanismo de segurança vigente.

Como a abordagem de *hooking* usada substitui o endereço de chamada de uma *syscall* por uma função existente do módulo, foi necessário desenvolver funções preâmbulas para

este fim. Desta forma, o OSPI só é capaz de controlar *syscalls* previamente conhecidas e implementadas.

Dado um processo qualquer que faça uso de uma *syscall* que está sob controle do OSPI e um objeto acessado que seja parâmetro desta *syscall*, o fluxo de execução ocorre da seguinte forma (internamente à função *Hooked Sys N*):

1. A dupla *process id (PID)* e *object address* (*e.g.*, endereço do *file descriptor*) é passada ao gerente de acesso para obtenção das permissões de acesso necessárias.
2. O gerente de acesso consulta se existe uma entrada na tabela *hash* referente ao objeto e, caso exista, a qual nó da lista de processos ela pertence.
3. Caso o objeto não seja controlado, o acesso continua para a *syscall N* como usual.
4. Caso o objeto seja controlado, o nó do processo que detém as permissões do objeto é obtido.
   (a) Se *PID* encontra-se na fila de processos ou em um grupo controlado, a máscara de permissões é retornada.
   (b) Caso contrário, o erro `EACCES/13` ou *Permission denied*, é retornado ao processo.
5. Com a máscara de permissões, o gerente de acesso define qual modo de operação do OSPI será utilizado, se o processo possui a posse e como o acesso irá ocorrer.
6. De posse das permissões de acesso, a função *Hooked Sys N* aplica as operações de acesso, como o uso de criptografia, antes de passar a execução para a *syscall N*.
7. Caso o acesso não deva ocorrer, o erro `EACCES/13` é retornado ao processo.
8. O retorno da *syscall N* é tratado pelo OSPI e, se necessário, outras operações ocorrem, como a decifragem de dados lidos pela *syscall*.
9. Por fim o processo recebe o retorno da *syscall*.

4.1.1 Syscall hooking

O *kernel Linux* mantém uma tabela de apontadores, que fazem referência às chamadas de sistemas ou *system calls*, tornando possível ao nível de usuário realizar operações com os privilégios do *kernel*. Essa tabela de apontadores é chamada de *syscall table* e em conjunto com as *syscalls*, formam a *sycall layer* do *kernel* ou também *syscall subsystem*.

O conceito de subsistemas no *kernel Linux* é utilizado como uma maneira de separação lógica das áreas de memórias do *kernel*. Tal separação visa tanto a contenção de falhas, como uma negação de serviço atingir um subsistema e não todo o *kernel*, quanto a otimização do código. Um exemplo dessa otimização (consequência da separação entre subsistemas) é a tabela de símbolos do *kernel*, pois os símbolos de um subsistema não estão presentes em um outro subsistema, reduzindo o tempo de acesso e o tamanho necessário para cada tabela de símbolos. Porém, essa abordagem gera um problema ao OSPI:

o subsistema de módulos não tem acesso, e de fato não deveria ter, à tabela de *syscalls*. A tabela de *syscalls* deveria ser manipulada exclusivamente no processo de *boot*, no *syscall subsystem*, para o mapeamento das *syscalls* compiladas e existentes no executável do *kernel*, em seus apontadores em memória. Além do símbolo da tabela de *syscall* não estar disponível em outros subsistemas, a região de memória onde essa tabela reside, desde a versão *2.6.12* do *kernel*, é marcada como *read-only*.

Nas arquiteturas *i386* e *x86-64*, a proteção das áreas de memória do *kernel* são controladas pelos registradores *Control register (CR0-CR4)* do processador. Ao final do *boot* do *kernel*, assim que o subsistema de *syscalls* termina o carregamento e antes do subsistema de módulos iniciar, o *kernel* trocar o registrador `CR0` para *write-protect*, evitando que a *CPU* consiga escrever em áreas de memória marcadas como leitura. Devido a essas limitações, o mecanismo de *hooking* foi dividido nas seguintes etapas:

1. Localizar, na memória do *kernel*, a tabela de *syscall*.
2. Localizar o deslocamento do apontador da tabela de *syscall*, para a função que executa a *syscall* do *kernel*.
3. Marcar o segmento de memória que contém a tabela, como gravável.
4. Substituir o apontador para a *syscall* na tabela, para a implementação da syscall controlada pelo OSPI.
5. Marcar o segmento de memória da tabela, como somente leitura novamente.

A abordagem para localizar a tabela de *syscall* escolhida foi aquela que usa a varredura da memória principal. Apesar de existirem outras abordagens, como o *parsing* de `/boot/System.map-versão-kernel` que contém todos os símbolos do *kernel* para *debug*, a varredura da memória pode ser feita independentemente da maneira que o *kernel* foi compilado, incluindo a existência ou não de *features* de *debug*.

A varredura simples consiste no conhecimento do endereço de uma função *syscall*, como `sys_open`, na leitura de *4-8 bytes* de uma seção de memória por vez e na comparação dos *bytes* lidos, como apontadores e o endereço da *syscall*. Caso coincidam, os *4-8 bytes* posteriores devem coincidir também com a função `sys_close` e depois `sys_access`. Os dois testes garantiriam que a leitura, com alta probabilidade, estaria em cima da tabela de *syscall*, estando o início dessa tabela no apontador referente aos cinco apontadores anteriores. Entretanto, essa abordagem é computacionalmente custosa, principalmente a varredura de longos blocos de memória. Uma abordagem mais sofisticada utiliza o conhecimento prévio de que nenhuma função no *kernel* Linux é diretamente apontada, haja vista a necessidade de ajustes e otimizações entre arquiteturas diferentes. Ou seja, o endereço de memória que armazena o apontador referente a chamada de `sys_close`, se usado como apontador de apontador em um interação, possivelmente estará iterando na própria tabela de *syscall*. O trecho de código a seguir exemplifica tal iteração—na linha 8, `__NR_close` é o índice da *syscall* `close` na tabela.

```
1 unsigned long ptr_func;
2 unsigned long *ptr_systable;
```

```
3  for (ptr_func = (unsigned long)sys_close;
4       ptr_func < (unsigned long)&loops_per_jiffy;
5       ptr_func += sizeof(void *))
6    {
7        ptr_systable = (unsigned long *)ptr_func;
8        if (ptr_systable[__NR_close] == (unsigned long )sys_close)
9          return (unsigned long **)ptr_systable;
10       else
11         printk(KERN_ERR "(OSPI): Can't find the sys_call_table!!\n");
12    }
```

Ao final da varredura daquela região de memória, a tabela de *syscalls* é encontrada. A próxima etapa para o *hooking* é a escrita do endereço da função OSPI para gerenciar a referida *syscall* nessa tabela, na posição da função que receberá o *hook*. Para efetuar essa alteração, deve-se tornar a região de memória da tabela de *syscall* como passível a escrita. Isso é feito com a escrita do *bit* "0" na posição de *bit* 16 do registrador `CR0`.

O *kernel* possui alguns símbolos para operações no registrador: `CR0_WP` que é uma palavra do processador com o 16º *bit* como "1"; `write_cr0` função que encapsula as operações dependentes de arquitetura para escrita diretamente no registrador; `read_cr0` encapsula operações para a leitura da palavra no registrador. As seguintes linhas de código representam o processo da troca de permissão da tabela de *syscall* e da escrita da *syscall* gerente do OSPI. O vetores `orig_sys` e `ospi_sys`, armazenam apontadores para a *syscall* do *kernel* e do OSPI, respectivamente. Nas linhas 8 e 9, o índice `NR_sys` é um inteiro referente a posição da *syscall* a sofrer *hook* na tabela.

```
1  cr0 = read_cr0( );
2  write_cr0(cr0 & ~CR0_WP);
3
4  addr = (unsigned long) syscall_table;
5  ret = set_memory_rw(PAGE_ALIGN(addr) - PAGE_SIZE, 3);
6   if(ret)
7       return -1;
8  orig_sys[k] = syscall_table[NR_sys];
9  syscall_table[NR_sys] = ospi_sys[k];
10
11 write_cr0( cr0 );
12
13 return 0;
```

4.1.2 Modos de operação

Dentro do *hook* de uma *syscall*, a gerência de acesso do módulo pode atuar de duas maneiras distintas: *managed* ou *restricted*. No modo *managed*, as políticas são aplicadas de acordo com a máscara de permissões, ao conceder ou negar acesso aos objetos controlados, retornando um erro na *syscall* quando negado. Nesse modo também é possível o uso de um objeto efêmero, criado exclusivamente como um objeto virtual. Seguindo a máscara de permissões, um processo pode ter todas as suas gravações confinados ao sistema de

arquivos virtual, em se tratando de arquivos ou *buffers* geridos pelo *kernel*. A execução efêmera permite que um processo sustente uma execução privada de forma transparente, de modo que todas as alterações são confinadas na memória do *kernel* até o fim da execução, com o processo alheio a tal controle. Esse mecanismo de *I/O* foi feito nas *syscalls* `write` e `read`, onde cada *file descriptor* foi mapeado como um *buffer* dinâmico de páginas de memória do *kernel*.

O modo *restricted* garante a privacidade e segurança de uma forma mais intrusiva, por meio da cifragem dos dados ao escrever e decifragem ao ler, de todos os objetos tocados por um determinado processo. Isso impede que outros processos, independentemente de sua permissão, consigam ler dados em texto claro de um processo executado no modo restrito. Este modo pode ser persistente, de maneira que o processo ainda possa ler os dados manipulados mesmo se o sistema for reiniciado. Caso a persistência dos dados faça-se necessária, os diferentes processos lançados pelo mesmo executável terão a mesma chave de criptografia, invalidando assim o requisito que garante o isolamento de dados mesmo para diferentes instâncias do mesmo executável.

A cifragem e decifragem dos dados ocorrem em *syscalls* como `read` e `write` ou `msgget` e `msgsnd` (IPC), usando a *CriptoAPI* [25] do *kernel Linux*. Essa *API* suporta diversos criptossistemas e funções de *hashing*, utilizando sempre as implementações e abordagens dos algoritmos que melhor aproveitem os dispositivos de *hardware* presentes. As operações sobre os objetos controlados ocorrem utilizando uma chave no criptossitema *AES128* disponibilizado pelo *kernel*. Uma vez que tais operações são feitas internamente nas *syscalls* controladas, essas são transparentes para os processos envolvidos. A chave utilizada é composta de duas outras chaves—a chave do módulo, que nunca é armazenada na memória principal e a chave privada, única para cada processo controlado.

A geração da chave privada do processo pode ocorrer de duas formas, através do gerente em nível de usuário que devolve o *hash* SHA512 do *ELF* relativo ao executável do processo, ou através do módulo utilizando informações da *task_struct* do processo. O uso do SHA512 foi feito através da truncagem por *XoR (eXclusive OR)* para 128 bits (para uso com AES128), utilizando-se o *Linux CryptoAPI* para tal operação. Referências como [27] e [12] corroboram que a truncagem não enfraque um hash criptográfico considerado seguro, desde que os requisitos da entrada se mantenham válidos. Dado que o módulo desenvolvido é um protótipo para prova de conceito, a gerência em nível de usuário funciona para quaisquer *kernels* da família 3 e 4, enquanto a gerência exclusiva via módulo funciona apenas na família 3 do *kernel Linux*. Alterações na exportação de símbolos na família 4, como no *procfs* ou no escalonador, geraram problemas na compilação do módulo (devido principalmente a *features* como *live patching* e *DAX* [6]). O módulo do OSPI utiliza um tipo `mm_segment_t` da estrutura `thread_info` e um tipo `mm_struct` da estrutura `task_struct` para obter a seção *.text* do processo e assim calcular o seu *hash* SHA512. Um detalhe importante referente à implementação é que o endereço apontado pela variável `mm_segment_t` deve ser convertido para o endereço real, dado que a seção *.text* começaria no endereço virtual *0x00000000*. Essa conversão é feita pela tupla de funções do *kernel* `get_fs` e `set_fs`.

Independente da abordagem utilizada, o *hash* é capaz de representar todo o código disponível a ser executado pela *task* e, com alto grau de confiança, ser a chave única

de um processo controlado mesmo quando a máquina reiniciar e todos os identificadores do processo mudarem. Caso a execução não tenha o bit de persistência, uma semente pseudo-aleatória provida pela entropia do *kernel* é adicionada como *salt* ao *hash*, tornando possível que processos de um mesmo executável mantenham instâncias privadas entre si. Uma vez que uma execução não-persistente termine, todos os objetos manipulados pelo processo em questão podem ser considerados perdidos, em vista da criptografia utilizada e da perda da chave.

A chave privada do processo não pode ser usada diretamente como a chave no criptossistema *AES128*, pois não existe a confidencialidade desta, uma vez que o hash do processo pode ser replicado por um atacante. Desta forma, a chave utilizada é o resultado da operação *XOR* (ou-exclusivo) das chaves privadas do processo e do módulo. O uso de tal operação, para a geração de chaves únicas a partir de uma chave *tamper-proof* (chave do OSPI) e outras sem tal garantia(chaves dos processos), não enfraquece ou expõe a primeira chave, ao passo que torna as chaves geradas diferentes entre si, tanto quanto as chaves dos processos são diferentes. *Graham Steel* [75] demonstra como podem ocorrer ataques a partir da extração de características das chaves geradas por meio de algoritmos que usam operações *XOR* em *Security APIs* largamente utilizadas na infraestrutura bancária. Portanto, a segurança contra a dedução da chave gerada no presente trabalho depende da segurança do algoritmo utilizado para o cálculo do *hash* e do algoritmo que gerou a chave do módulo. Tal segurança pode ser violada se houver preservação mútua das características presentes nas saídas dos algoritmos ou se as características da entrada forem preservadas na saída.

O estudo supracitado apresenta outra fragilidade adicional à extração de características: a propriedade do inverso, na qual o resultado da operação será sempre "0" (zero). Esse ataque depende do conhecimento da chave resultante e de uma das chaves da composição, no caso a chave privada de um processo, em vista da facilidade de calcula-la. Porém, a chave do processo sozinha não tem a intenção de ser sigilosa e de fato pode ser conhecida, dado que a chave resultante e a chave do módulo nunca residem na memória principal. Logo, ambas as chaves só podem ser conhecidas em nível de *kernel* por meio da leitura de registradores. A chave resultante é produto de instruções *AES-NI* do processador e, portanto, é armazenada em registradores. Essa chave imediatamente após ser gerada serve como parâmetro no *AES128* através de instruções AES-NI (*Advanced Encryption Standard New Instructions*) presentes no processador. Isso faz com que ela não seja armazenada na memória principal. Para que isso ocorresse, foi necessário o uso diretamente das instruções citadas, dado que o uso das funções de cifra da *Crypto API* colocaria a chave na memória principal, antes de movê-la para os devidos registradores.

A chave do OSPI é utilizada para todos os processos, mas não deve ser gerada pelo módulo e sim por uma boa fonte de entropia. Pretende-se, em uma versão posterior, que o módulo seja capaz de ler a chave nas primeiras fases do *boot*—por meio de um dispositivo de armazenamento de confiança ou um *hardware* gerador—e, após a leitura preencha com "0" cada *buffer* usado para transporte da chave, armazenando-a, por fim, nos registradores de *debug* do processador (`DR0` a `DR3`). Esses registradores não podem ser utilizados fora do modo privilegiado do processador, a não ser que um *debugger* executando como *root* os acesse via capacidades especiais providas pelo *kernel Linux*.

O OSPI usa diretamente as funções do *kernel* para desabilitar e impedir o uso dos registradores de *debug* por meio das seguintes funções: `arch_uninstall_hw_breakpoint`; `hw_breakpoint_restore`; `arch_install_hw_breakpoint` (irá sofrer *hook* e chamar `hw_breakpoint_restore`). Desta forma, inviabiliza-se o uso de *hardware breakpoints* ao mesmo tempo em que se aumenta o nível de segurança do sistema, pois há trabalhos que mostram como *rootkits* abusam dessa funcionalidade de forma a persistir no sistema alvo. Com isso, tais *rootkits* podem comprometer tanto o OSPI quanto o *Grsecurity* [10] [53].

A Figura 4.5 ilustra as interações entre os diversos níveis de processos presentes em um sistema. As interações representadas pelas arestas pretas pertencem a processos não-controlados, enquanto que as policromáticas representam os processos sujeitos a alguma forma de controle. Arestas azuis ou verdes são interações com objetos pertencentes aos grupos "0" ou "1", cujos dados são cifrados ou decifrados em decorrência do parâmetro presente em uma *syscall* controlada. As interações negadas são representadas por arestas amarelas, nas quais os processos não possuem entrada na lista de processos controlados e, portanto, não possuem uma chave simétrica.

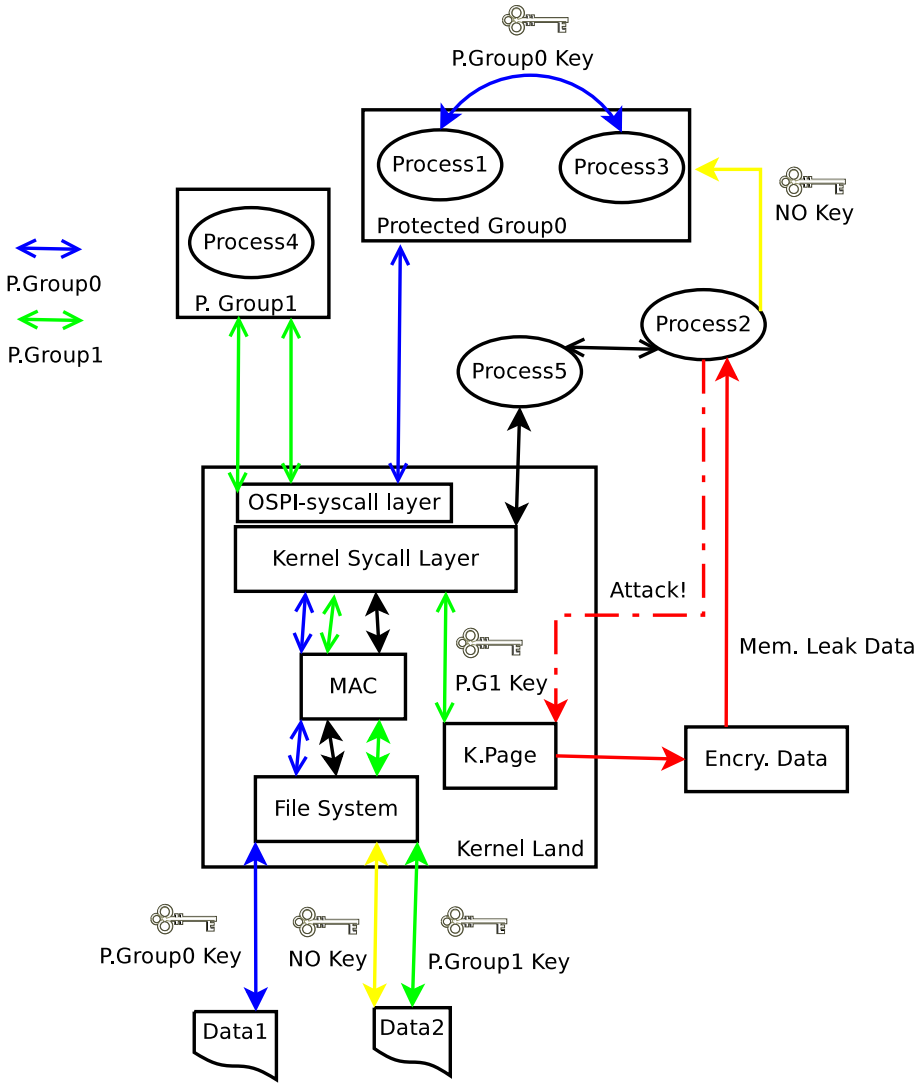


Figura 4.5: Diagrama de interações entre grupos de processos controlados (setas coloridas) e suas chaves com processos fora do escopo de controle do OSPI (setas pretas).

Caso processos do *grupo0* tentem acessar objetos do *grupo1*, mesmo sem uma regra no *MAC* para coibir essa ação, o objeto é decifrado com chave errada, não revelando assim suas informações. As arestas vermelhas indicam uma interação anômala, enquanto que a aresta tracejada indica um ataque não-detectado pelo *Grsecurity*. Mesmo que esse vetor de ataque ocorra com sucesso, a leitura dos dados não terá utilidade, uma vez que o OSPI cifrou o conteúdo da memória vazada, fazendo com que o processo atacante não consiga recuperar os dados em claro.

4.2 Limitações e superfície de exploração

Como todo mecanismo de segurança, o OSPI possui limitações. Ele foi proposto como um mecanismo de proteção a ser usado em conjunto do *Grsecurity*, principalmente para sanar as limitações desse mecanismo de segurança, além de também fazer uso de sua proteção. Essa abordagem protege o OSPI de diversos cenários de ataques, sem a necessidade da construção de um mecanismo de segurança completamente independente para o *kernel* Linux. Assim, apesar das limitações quanto a implementação do módulo, o OSPI resiste a ataques contra o *kernel* que possivelmente abusariam de seu mecanismo de *hooking*, como ataques de força bruta em descritores de arquivos, caso o *Grsecurity* não estivesse presente.

Fazendo do uso da interface modular do Linux, o OSPI assim como qualquer módulo, somente é carregado após o *kernel Linux* habilitar o nível de usuário, estando assim no mesmo nível hierárquico de outros módulos. Visando reduzir a superfície de ataques, o módulo foi incluído dentro do *initramfs*, a fim de ser carregado nos primeiros estágios do *boot* do *core* do *kernel*. Desta forma, o OSPI tem garantias de sempre ter seu carregamento efetuado, independente da existências de outros módulos (como alguns *rootkits*). Porém, mesmo que o OSPI proteja-se contra alterações no *initramfs* enquanto ele estiver carregado, ele não pode se proteger contra alterações nos parâmetros de *boot* (geridos pelo *bootloader*), tanto de usuários que tenham acesso físico a máquina, quanto a *bootkits*[18].

De qualquer forma, como o principal mecanismo do OSPI é a interposição de *syscalls*, ele está teoricamente sujeito a diversos ataques de *race condition*. *Garfinkel* [16] resume os principais problemas em mecanismos de interposição de *syscall*, principalmente em relação a falhas de *race conditions*, como a *TOCTOU (time of check to time of use)*. O *TOCTOU* é uma classe de erro em *software*, causado por alterações em um sistema entre a verificação de uma condição (tal como uma credencial de segurança) e a utilização dos resultados dessa verificação. Especificamente em mecanismos de interposição de *syscalls*, esse erro ocorre quando um grupo de processos utilizam tanto a *syscall* do *kernel* quanto a *syscall* de interposição, mesmo após a *syscall* ter sofrido corretamente o *hooking*. O OSPI foi feito com o conhecimento dessa classe de falhas e possui alguns mecanismos para evita-la ou reduzi-la a ponto da segurança contra *kernel brute-forcing* do *Grsecurity* reduzir as chances de sucesso dramaticamente. Mesmo sem a garantia do *Grsecurity*, a correta construção dos mecanismos de segurança é dada como suficiente para evitar *race conditions*. Trabalhos como [59] e [3] indicam um crescente uso desse mecanismo de interposição de *syscall*, considerando-se o uso correto das estruturas de controle, evitando

assim *race conditions*.

Apesar do OSPI ter sido construído com essa classe de falhas em mente, existem casos teóricos onde é possível haver *race conditions*. Logo, as seguintes limitações foram impostas para reduzir a superfície de ataques possíveis:

- Todas as *syscalls* a serem controladas sofrem *hooking* antes da interface ao *userland* estar disponível.
- Toda operação das *tasks* no módulo segue a *kernel queue* do módulo, com o uso de *mutex locks* para garantir a política de *FIFO* no acesso aos objetos, uma vez que as *syscalls* devem ser reentrantes.
- A interface do *procfs* utiliza todas as funções de *callback* do *kernel*, onde o módulo lê o bloco de dados resultante. Isso garante que toda *userland I/O* é devidamente tratada pelas políticas-padrão do *kernel*, reforçadas pelo *Grsecurity*, antes de chegarem ao OSPI.
- Toda operação em grupos utiliza um *mutex* exclusivo ao grupo, evitando que a máscara mude após uma leitura parcial ou que um processo saia durante uma operação (*e.g., IPC pipe*).
- As operações no *restricted mode* são exclusivamente bloqueantes e a *syscall open* sempre utiliza a `flag O_DIRECT` (*I/O* sem o *kernel buffer cache*).

Essas limitações, em grande parte, são devidas as restrições quanto a implementação de um mecanismo mais complexo para a gerência concorrente e escalável de *syscalls*, sendo assim uma limitação técnica na interface com a camada de *syscall* no *kernel Linux* ao implementar e administrar a concorrência. Também há invalidação do cache da CPU, ao se fazer o *switch* de uma *task* entre *cores*, já que o módulo reside em um *core* fixo.

Entretanto, tais limitações evitam a existência de *race conditions* ao longo do fluxo de execução do OSPI, o que foi comprovado através de *syscall fuzzing* (descrito no Capítulo 5). A aplicação de *hooking* anterior à existência de processos controlados também resolve outra limitação: a construção de um mecanismo genérico de *hook* para qualquer *syscall*, baseado na assinatura da função da memória. A predefinição em tempo de compilação de todas as funções gerentes de *syscalls* e seus protótipos similares as *syscalls* do *kernel* simplifica o processo de *hooking*, apesar de limitar a proteção para uma lista fechada de *syscalls*. É possível se construir um mecanismo genérico ou se adicionar a maioria das *syscalls* do Linux para que fiquem sob controle do OSPI. Porém, em vista do projeto como uma prova de conceito, isto é, mais teórica que tecnológica, uma abordagem mais específica foi utilizada para validar o que foi proposto nesta dissertação.

4.2.1 Explorações

Tendo em vistas as limitações tecnológicas, o OSPI possui uma falha apta a exploração, inerente a sua abordagem de controle. Ao utilizar o endereço dos identificadores de objetos a serem controlados, há a possibilidade de dois objetos, cujos identificadores sejam

diferentes, serem de fato o mesmo objeto no *endpoint* do *kernel*, tal como no *HAL (Hardware Abstract Layer)*. Um exemplo disso é um arquivo e seu *hardlink* possuírem dois identificadores diferentes de arquivo, por terem *path* diferentes. Essa falha está presente também no *AppArmor*, porém no OSPI uma solução intermediária foi aplicada. Apesar de não haver como evitar esse comportamento, caso existam *hardlinks* anteriores à execução do módulo, é possível forçar na *syscall open* a *flag* `O_TMPFILE`. Essa *flag* faz com que, quando o arquivo da *syscall* deva ser criado, ele seja seguro contra a criação de *hardlinks*, mitigando parcialmente a referida falha.

Além da exploração inerente do OSPI, existem vetores de ataques possíveis em quase todos os subsistemas do *kernel*: *overflow* ou *brute force* de apontadores ou *tokens*. Como qualquer usuário do sistema pode ter acesso a interface com o OSPI, é possível que um atacante utilize o *brute force* de *PIDs* e máscaras de bits, até que ele acerte um processo a ser protegido, dentro da janela de configuração. Esse ataque somente é mitigado com a presença do *Grsecurity*, capaz de detectar e evitar tanto o *brute force*, quanto o acesso no *procfs* por entradas que pertencem ao processo atual, logo evitando o acerto ocasional da referência alheia.

Capítulo 5

Testes e Resultados

Neste capítulo, são apresentados os testes empíricos realizados para comparar o mecanismo desenvolvido com os outros mecanismos estudados, bem como os resultados obtidos. Os testes de eficácia foram feitos por meio do uso de *exploits* em falhas presentes nas aplicações e no *kernel* do ambiente testado. Com isso, validou-se a premissa de acréscimo de segurança pela implantação do OSPI. Foram também realizado testes baseados em *fuzzing* para se medir o *overhead* do OSPI, assim como sua estabilidade ao ser adicionado no nível do *kernel*.

5.1 Falhas e exploits

O desenvolvimento do módulo de *kernel* proposto foi feito almejando um mecanismo com fina granularidade e proteção em baixo nível de forma a satisfazer as necessidades de segurança de processos sensíveis ou especiais. O mecanismo OSPI foi projetado para ser capaz de aumentar o nível de privacidade dos dados nos casos onde os mecanismos de proteção existentes falham. Além disso, a combinação da abordagem proposta com o *Grsecurity* faz com que um determinado processo possa ser protegido contra violações de privacidade e ataques em cenários incomuns, como mostrado adiante no capítulo.

Todos os testes ocorreram em dois ambientes distintos, um virtualizado e outro *bare metal*. Os testes de *syscall fuzzing* e com *exploits* foram executados em ambiente virtualizado com *qemu-x86_64 2.0.0+dfsg-2*. A imagem utilizada foi a disponível em [26], com *updates* seletivos, via *apt-get*, para as vulnerabilidades testadas. O sistema hospedeiro faz parte da infraestrutura do LASCA[1] no Instituto de Computação da Unicamp, sendo este um *Gentoo* com *kernel Linux 4.4.6* em máquina com processador *Intel(R) Xeon(R) CPU E5630 @ 2.53GHz*. Já os testes em *bare metal* foram feitos em máquina com processador *Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz* com sistema operacional *Debian 7*, *kernels 3.14*, *3.18* e *4.6*. O uso de diversos *kernels* em *bare metal* ocorreu para o teste seletivo dos subsistemas do OSPI, assim como das vulnerabilidades e versões do *Grsecurity*. As versões testadas do *Grsecurity* foram todas *test/unstable*, umas vez que esse mecanismo encerrou o acesso livre/gratuito à versão estável em 9 de Setembro de 2015.

Para validar o protótipo desenvolvido em sua versão mais atual, bem como a verificar

[1]Laboratório de Segurança e Criptografia Aplicada

a atuação dos outros mecanismos de segurança disponíveis, utilizou-se de alguns ataques recentes que não foram bloqueados, sem novas regras ou configuração, pelos mecanismos estudados. O primeiro ataque a ser testado foi o de exploração da virtualização em *kernel* do *Docker*: *docker container breakout* [84]. Até ser publicamente conhecido, o ataque não possuía uma regra de detecção correspondente no *Grsecurity* nem no *AppArmor* ou *SELinux*. Assim, esses mecanismos não eram capazes de evitar que o ataque fosse completado com sucesso. A falha em questão contou com uma combinação de *syscalls* e *kernel capabilities* que não estavam na *blacklist* de nenhuma mecanismo de proteção até então. Por padrão, o *Docker* utiliza o *AppArmor* como seu mecanismo de política e segurança e, até a versão do *Docker* 0.9, o *profile* do *AppArmor* se resumia às restrições aos *namespaces* de *PID*, *IPC*, *mount points*, em alguns casos *network*, além da *blacklist* para *capabilities*.

Essa falha no isolamento do *Docker* necessitava que, internamente a um *dock* (contêiner virtualizado), uma *task* acessasse a camada de *syscalls* com *uid* 0 ou superusuário. Apesar do isolamento do processo que executasse esse *dock* o separasse do sistema hospedeiro (*e.g.*, visão independente de dispositivos, processos, sistema de arquivo, rede e *hardware*), o acesso a camada de *syscall* é feito diretamente, pois trata-se do mesmo *kernel*. Apesar de não ser recomendado executar *tasks* como *root* em um *dock*, nenhum objeto fora do isolamento do *Docker* deveria ser acessível, dadas as visões diferentes do sistema, como os *namespaces* do *kernel*. Entretanto, o acesso com *uid* 0 nas *syscalls* e o uso da tupla de *syscalls* *name_to_handle_at()* e *open_by_handle_at()*, tornou possível a um atacante abrir um arquivo fora do isolamento, dado que ele possuísse seu *handler* e o arquivo estivesse previamente aberto por outro processo.

Como o contêiner compartilha, via *task_struct*, diversas estruturas com os processos de fora e o conhecimento prévio do atacante de que o *inode* 2 é a raiz do sistema de arquivo, foi possível utilizar *name_to_handle_at()* partindo da raiz, até obter um *handle* válido e usar *open_by_handle_at()* para abri-lo. Esse *handle* é um vetor de 64 *bits*, onde os últimos 32 *bits* indexam o número do *inode*. Com a raiz sempre no *inode* 2, essa busca envolveu a força bruta de 32 *bits*, até encontrar um *handle* válido ou um válido e específico, como para */etc/shadow* ou */etc/sudoers*. Para que o uso dessas *syscalls* em *handles* que não pertencem ao usuário do *dock* fosse possível, foi preciso haver duas *kernel capabilities* disponíveis: *CAP_DAC_READ_SEARCH* e *CAP_DAC_OVERRIDE*. A primeira permite que o *root* consiga sobrepor a checagem de permissões na leitura e execução em diretórios e arquivos, e a segunda na sobreposição da leitura, escrita e execução de arquivos. Isso possibilita que um *dock* consiga manipular arquivos fora de seu isolamento.

O motivo do sucesso deste ataque em mecanismos como *Grsecuriy* e *AppArmor* é o fato de que, mesmo que o processo isolado não seja executado como *root* internamente ao *Docker*, um processo com privilégio de *root* executa *syscalls* como tal. Isto ocorre devido a virtualização em espaço de usuário compartilhar a mesma *syscall layer* do resto do sistema. Para barrar esse ataque, *AppArmor* recebeu um novo *profile* em que todas as *capabilities* foram bloqueadas por padrão (exceto para aqueles em uma *whitelist*).

O OSPI foi capaz de bloquear o ataque sem sequer conhecê-lo previamente, com o uso do modo *restricted*. Uma vez que um *dock* esteja nesse modo de proteção, toda operação

de leitura é feita utilizando, internamente a *syscall read*, a chave simétrica para decifrar o conteúdo da lido (que foi cifrado na escrita). Logo, caso ocorra uma quebra na contenção do *dock*, este ao realizar a leitura fora do isolamento obterá dados inteligíveis, uma vez que dados não cifrados ao serem submetidos às rodadas de decifragem do *AES*, resulta em uma saída similar a cifragem. O bloqueio indireto do ataque pelo OSPI ocorreu de forma similar em relação à falha do *Sudo (CVE-2015-5602)* [77]. Essa falha explora o mecanismo de resolução de expressões regulares do *sudo* e das resoluções de permissões de *symbolic links*, permitindo ler arquivos arbitrários com permissões de superusuário. Assim, de maneira similar ao *Docker*, o processo susceptível ao ataque quando colocado no modo *restricted*, terá como resultado dados inteligíveis na leitura, quando feita em arquivos diferentes daqueles escritos pelo próprio processo.

Portanto, o módulo proposto já é capaz de atenuar uma classe inteira de ataques semelhantes somente com a utilização do modo restrito no processo protegido. Assim, um processo que quebre o isolamento provido pelo *Grsecurity* não será capaz de ler qualquer objeto que não seja seu, de acordo com a política de acesso do *OSPI*. Mesmo que o processo consiga subverter as restrições do *MAC* e ter acesso ao objeto, dado que o conteúdo do objeto está cifrado, esse acesso não será bem sucedido no contexto de um ataque de violação da confidencialidade. O cenário da escrita funciona de forma semelhante: como o processo que possui a posse do objeto sempre o recebe na leitura após o modulo decifra-lo, a escrita direta deste objeto por um ataque resultará em dados ininteligíveis para o processo protegido. Desta forma, o processo subvertido não poderá escrever o que pretende, pois mesmo que haja alteração dos dados, esta corrompe o arquivo, mitigando o ataque (por exemplo, a escrita de credenciais em */etc/shadow* resultaria em uma linha inválida e, portanto, inútil para um *login* do atacante).

Apesar de não ter sido um dos objetivos do presente módulo, durante a fase de testes por meio da execução de *exploits* e *rootkits*, notou-se que o OSPI é capaz de detectar ou até mesmo evitar a implantação com sucesso de *rootkits* que fazem uso de *syscall hooking*. A detecção ocorreu quando um *watchdog* do OSPI, inicialmente usado na gerência de concorrência, emitiu alertas (via *system log*) que a região de memória da *syscall table* tornou-se *writable* inesperadamente (ocasião da instalação de *rootkit*). Como o módulo foi feito para ser carregado durante os primeiros estágios do *boot* através da composição do *initramfs* para inclui-lo, o OSPI presumidamente será o primeiro a efetuar o *hooking* na tabela de *syscall*. A fim de evitar manipulações nocivas ao *initramfs*, as quais influenciariam na ordem de carregamento dos *kernel modules*, esse arquivo fica sob proteção do modo *managed*, porém sem proprietário. Desta forma, a única maneira de manipular o carregamento do módulo seria por meio do modo de manutenção/recuperação do sistema, passando a superfície de ataque ao *GRUB*[2] em seu mecanismo de autenticação para escolha dos parâmetros de *boot* do *kernel*, ou ao *UEFI*[3] da placa mãe.

O *rootkit Diamorphine* [39] faz *syscall hooking*, além de interceptar sinais entre *user-kernel* para omitir módulos (incluindo o do próprio *rootkit*) na lista de módulos, bem como processos e arquivos. Ele também permite a elevação de privilégios via *kernel* de qualquer *task* que envie um sinal (*signal*), o qual é interceptado pelo *rootkit* para identificar a *task*

[2]GRand Unified Bootloader

[3]Unified Extensible Firmware Interface

a ser elevada. Esse *rootkit* foi instalado para observar o comportamento do sistema ao ser protegido pelo *Grsecurity* e pela combinação *Grsecurity + OSPI*. Uma vez que o módulo foi instalado, dado que o usuário não possuía restrições a essa operação, o *Grsecurity* não evitou os *hooks*, porém foi capaz de evitar a elevação de privilégios e a omissão do módulo *rootkit* da lista de módulos. Quando esse *rootkit* foi instalado no ambiente sob monitoração do OSPI, ele aplicou sua interposição de *syscall* antes das funções do OSPI, acreditando serem essas funções as *syscalls*. Nesse ambiente, o *rootkit* poderia manipular a entrada do usuário e assim obter seus dados sigilosos o que, mesmo não sendo parte de seu algoritmo, é uma ameaça em potencial. Porém, como o OSPI continua fazendo a interface com as *syscalls* do *kernel*, ele pode negar o acesso a objetos deste, como a lista de módulos carregados.

Com os resultados desse teste empírico, uma abordagem experimental foi feita para bloquear completamente o ataque por *syscall hooking*: um *watchdog* foi instanciado para monitorar a tabela de *syscalls*. Essa abordagem também está presente nas últimas versões do ano de 2016 no *Grsecurity*, junto com a proteção contra *ROP (Return Oriented Programing)* [58]. Porém, dado que essas versões são exclusivamente para os *sponsors* daquele mecanismo de segurança, a abordagem presente no OSPI pode ser de grande valia.

5.2 Comparação de *overhead* com outros mecanismos

Embora o *OSPI* ainda não tenha sido exaustivamente testado, o sucesso alcançado no bloqueio de alguns ataques que não são tratados por outros mecanismos de seguraça sem configurações muito específicas mostra que a abordagem é promissora. Os cenários avaliados—*rootkits (debug registers)* e *docker leak*—valida sua eficácia como um mecanismo complementar para reforçar a segurança de um sistema operacional e seus processos cuja proteção depende dos *frameworks* de segurança tradicionais.

Ao longo do projeto, foram feitas aferições acerca do *overhead* resultante dos mecanismos utilizados pelo módulo, a fim de se escolher a abordagem mais eficiente que, ao mesmo tempo, não implicaria em redução na segurança do sistema. As abordagens implementadas, tais como utilizar a fila de processos e objetos estruturadas em uma tabela *hash* e realizar a conversão dos endereços virtuais para os reais na *OSPI syscall layer* previamente à chamada original do *kernel*, reduziram muito o *overhead* e o *turnaround time*[4] das chamadas de sistema efetuadas. A Tabela 5.1 ilustra algumas *syscalls* executadas no mesmo *kernel*, porém com diferentes mecanismos de segurança instalados. O *turnaround time* de uma *syscall* é representado em milissegundos, enquanto que o *overhead* é em porcentagem da chamada em relação ao *kernel* sem nenhuma proteção. Foi utilizado o `SystemTap` e uma adaptação do *script iotime.stp* [57] para medição do tempo gasto nas chamadas.

Foram feitas 1024 chamadas de cada uma das *syscalls* representadas na Tabela 5.1, utilizando-se *1KB* de dados por meio da ferramenta *syscall fuzzer Trinity* [28] quando necessário. A chamada `null I/O` corresponde às execuções das *syscalls* `read` e `write`,

[4]Tempo necessário para completar um processo, do lançamento para execução até o retorno de sua saída.

porém com o *buffer* vazio. Todas as chamadas realizadas foram feitas de modo pseudo-aleatório, sempre seguindo a semântica dos parâmetros necessários para cada *syscall*. Com isso, elas representam tanto o comportamento esperado quanto o inesperado, simulando uma anomalia que possivelmente é tratada pelos mecanismos de segurança testados.

Tabela 5.1: Tabela comparativa com o *turnaround time* em milissegundos das *syscalls* em um sistema protegido pelos mecanismos OSPI, Grsecurity ou AppArmor em comparação à chamada sem proteção (base). As medidas de *overhead* correspondem à porcentagem do *turnaround time* da chamada protegida em relação à chamada desprotegida.

Syscall	Base	OSPI	Overhead	Grsecurity	Overhead	Apparmor	Overhead
`null I/O`	0,13	0,13	1%	0,13	5%	0,13	12%
`open/close`	2,05	2,05	4%	2,09	2%	4,28	109%
`fork`	131,17	131,43	2%	130,23	1%	134,71	27%
`execve`	386,52	575,91	49%	552,72	43%	587,51	52%
`pipe`	6,67	7,33	10%	7,67	15%	7,53	13%
`read`	8,38	11,98	43%	9,30	11%	9,14	9%
`write`	14,09	22,12	57%	15,49	10%	15,27	8%

Observa-se que o OSPI possui um *overhead* superior ao *Grsecurity* na maioria das chamadas. Isso era esperado, uma vez que enquanto o OSPI é um protótipo conceitual, enquanto o *Grsecurity* tem mais de 15 anos de desenvolvimento ativo, e assim possuiu um grande refino de seus mecanismos em nível de eficiência. Porém, com exclusões das *syscalls read/write*, onde ocorreram o uso do AES128, o OSPI manteve um *overhead* inferior ao *AppArmor*. Possivelmente isso deve-se ao fato que o *AppArmor* deve realizar o *parser* e a heurística do seu conjunto de regras e políticas, ao início de cada operação do sistema. Como essas atividades são bloqueantes, elas possivelmente contribuíram com o alto *overhead* desse mecanismo. As execuções das *syscalls* sem dados ou *null I/O*, comprovaram que a heurística de controle de acesso, a exclusão do criptossistema AES, teve um impacto pequeno no sistema. Uma vez que a não existência de dados como parâmetro da chamada, exigiu que o OSPI apenas varresse sua tabela *hash* para realizar o *matching* entre objeto e processo, o que de fato não é uma operação computacionalmente custosa.

Além do teste de *overhead*, o uso de *syscall fuzzing* também serviu como métrica de estabilidade nas estruturas internas e subsistemas do OSPI, em relação aos outros mecanismos testados. Como trata-se de um mecanismo conceitual, ainda em forma de protótipo, era esperado que houvesse problemas relativos a estabilidade. Das 1024 chamadas das *syscalls* controladas, durante os testes, 15.73% delas resultaram em *kernel OOPS*. Quando nenhuma *syscall* era controlada, 1.3% das chamadas tiveram aquele resultado. O *kernel OOPS* ocorre quando o *kernel Linux* encontra-se em um estado inconsistente (por exemplo, durante o acesso a uma região de memória sem permissão ou inválida), mas consegue recuperar-se sem negar a maioria dos serviços ao nível de usuário. Alguns desses *kernel OOPS*, cerca de 1.7% sem *syscalls* controladas e 3.8% no caso inverso, resultaram em um *kernel PANIC* iminente, que obstruiu o sistema de realizar qualquer outra operação, já que representa um erro crítico para o *kernel*. Sem o OSPI, apenas 0.5% das chamadas resultaram em *kernel OOPS* e não foram vistos *kernel PANICs*. Não houve

diferença nos testes quanto à estabilidade entre os outros mecanismos de segurança ou sem nenhum mecanismo de segurança presente.

Capítulo 6

Considerações Finais

Nesta dissertação, foram discutidos os mecanismos (ou *frameworks*) mais utilizados para a proteção da privacidade dos dados de processos em execução em sistemas operacionais Linux, mais especificamente, *Grsecurity* e *AppArmor*. Foram levantadas as limitações de tais *frameworks* em face de alguns ataques que exploram vulnerabilidades em programas cujo sucesso causa a exposição de informações sensíveis, seja por vazamento de dados ou pela presença de *malware* em nível privilegiado do sistema operacional. Com a finalidade de aumentar o nível de segurança do usuário desses programas, impedindo certas violações à confidencialidade, propôs-se o OSPI, um mecanismo de segurança que isola processos arbitrários do sistema, protegendo assim o contexto de seus dados. Apresentou-se a arquitetura do mecanismo proposto em detalhes, explicando-se seus componentes e procedimentos que alcançam o objetivo de melhorar o nível de privacidade dos usuários e seus processos. O mecanismo foi implementado como uma prova de conceito, a fim de que as premissas estabelecidas pudessem ser validadas.

Para validar a proposta, foram feitos testes empíricos que abrangeram as funcionalidades do OSPI, sua estabilidade e os resultados obtidos com sua execução. Com isso, foi possível notar que o mecanismo proposto alcançou, em média, um baixo *overhead* quando comparado com o *AppArmor*, mas obteve taxa de *overhead* superior ao *Grsecurity*. A diferença obtida em relação ao *AppArmor*, que possui em média um *overhead* inferior ao *SELinux*, foi notável, principalmente ao se considerar que as chamadas *open/read* apresentaram mais do que o dobro de *turnaround time*. Tal diferença é causada principalmente pelo tempo para se fazer o *parsing* dos arquivos de políticas, inexistente tanto no *OSPI* quanto no *Grsecurity*. Já o *overhead* ligeiramente superior, em relação ao *Grsecurity*, era esperado devido ao uso do criptossistema *AES* 128, que por si só demanda de um esforço computacional superior ao demandado pelo *Grsecurity*. Porém, esse acréscimo do esforço computacional é justificável pela proteção extra oferecida, onde o OSPI conseguiu frustrar ataques que, de uma maneira ou outra, passaram pelo *PaX* e *Grsecurity*. Isso foi devido ao mecanismo de gerencia de permissões e ao uso do *AES* 128 pelo OSPI, não dependerem do conhecimento prévio dos ataques, ou do comportamento do usuário, para aplicar suas políticas. Mesmo a implementação do OSPI sob a forma de protótipo, os resultados foram promissores e indicam o potencial da continuidade do projeto como um mecanismo viável de proteção complementar aos mecanismos existentes, suprindo uma lacuna de proteção mais granular.

Apesar dos problemas de estabilidade apresentados na Seção 5.2, como *kernel OOPS* e *PANIC*, pode-se considerar que o OSPI foi bem sucedido em atingir seu objetivo de proteger processos sensíveis contra ataques não considerados como tal pelos mecanismos de segurança disponíveis atualmente. Em vista da complexidade do desenvolvimento do módulo e da restrição de tempo para atingir os objetivos previstos para a conclusão do presente trabalho, muitas abordagens tiveram de ser minimalistas. O uso dessas abordagens limitaram a adição de novas funcionalidades ao OSPI, *e.g.* proteção ao *LSM* contra *rootkits*, a migração completa para a família 4 do *kernel* e a melhor resolução dos problemas de estabilidade. Porém, todas as escolhas foram feitas mantendo a segurança, sem comprometer a eficiência ou eficácia do sistema, e seguindo o lema do *Linus Torvalds* para *kernel-dev*, *KISS (Keep It Simple Stupid)*.

Durante a realização desta dissertação, foi publicado um artigo científico no principal simpósio nacional de redes de computadores, o SBRC [67]. Neste artigo, são apresentados os objetivos que motivaram o presente texto, a arquitetura proposta, a abordagem de proteção pensada para lidar com o problema do isolamento de processos arbitrários no *kernel Linux*, as limitações do mecanismo implementado como prova de conteito, bem como os testes preliminares de eficácia, eficiência e estabilidade realizados até o momento da aceitação para publicação.

6.1 Trabalhos Futuros

Diversas melhorias podem ser introduzidas para aprimorar a estabilidade do mecanismo, aumentar suas funcionalidades ou torná-lo mais genérico para inclusão e manipulação de *syscalls*. As principais possíveis melhorias são:

Assinatura de protótipo de função: Dado o número da *syscall* a ser interceptada, e conhecendo o protótipo dessa *syscall*, o OSPI poderia varrer o endereço de memória apontado pela *syscall table*, para fazer o *matching* da assinatura do protótipo da função a ser interceptada. Assim seria possível o OSPI interceptar as *syscall*, sem necessariamente construir uma função substituta para cada *syscall* a sofrer interceptação.

Sistema especialista: Com a adição do mecanismo de assinatura e protótipo de função, seria necessário a construção de um sistema especialista para a identificação do objeto a ser controlado na *syscall*. Esse sistema identificaria, entre os parâmetros da *syscall*, qual corresponde ao objeto (*e.g.* descritor de arquivo) a ser controlado. Atualmente essa identificação é feita manualmente em cada função do OSPI, que interpõem uma *syscall*.

Integração ao *kernel* 4: O OSPI apresentou problemas quanto a integração ao novo *kernel*, como resolução de símbolos e mudanças na interface e infraestrutura de alguns subsistemas (*e.g. procfs*), que inviabilizaram seu suporte na família 4 do *kernel Linux*. A reescrita dos códigos para o uso das novas interface, e a solução no uso, ou substituição, de símbolos não mais visíveis, permitiria a integração *kernel* 4.

Estabilidade: A maior parte dos problemas de estabilidades estão relacionados a concorrência, invalidação do cache, e principalmente uso indevido de alguma infraestrutura do *kernel*. A resolução desses problemas seria feita através do uso extensivo de *kernel*

debugging, como *kernel debugger internals* [30] e *Systemtap* [56]. Pois assim, seria possível monitorar o comportamento do OSPI ao manipular cada subsistema e estrutura do *kernel*, nas condições ainda não auditadas, onde as *syscalls* no teste de *fuzzing*, levaram o *kernel* a um estado inconsistente.

Referências Bibliográficas

[1] Martín Abadi. Logic in access control. In *Proceedings 18th Annual IEEE Symposium on Logic in Computer Science*, 2003.

[2] Ryan Ausanka-Crues. Methods for access control: Advances and limitations. Technical report, Harvey Mudd College, 2006.

[3] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '15, pages 27–38, New York, NY, USA, 2015. ACM.

[4] Neil Brown. Overlayfs documentation. `https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt`, 2015. Acessado em: 03 Abr. 2016.

[5] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Marco Squarcina. Gran: Model checking grsecurity rbac policies. In Stephen Chong, editor, *CSF*, pages 126–138. IEEE, 2012.

[6] Diego Calleja. Linux 4.0. `https://kernelnewbies.org/Linux_4.0`, 2015. Acessado em: 12 Jul. 2016.

[7] Kees Cook. Yama: a linux security module. `https://www.kernel.org/doc/Documentation/security/Yama.txt`, 2010. Acessado em: 30 Mar. 2016.

[8] NTT DATA Corporation. About tomoyo linux. `http://tomoyo.osdn.jp/about.html.en`, 2015. Acessado em: 30 Mar. 2016.

[9] Yu Ding, Tao Wei, Tielei Wang, Zhenkai Liang, and Wei Zou. Heap taichi: exploiting memory allocation granularity in heap-spraying attacks. In Carrie Gates, Michael Franz, and John P. McDermott, editors, *ACSAC*, pages 327–336. ACM, 2010.

[10] Joel Mattsson Emil Persson. Debug register rootkits. a study of malicious use of the ia-32 debug registers. Technical report, School of Computing,Blekinge Institute of Technology, 5 2012.

[11] David F. Ferraiolo and Richard D. Kuhn. Role-Based Access Controls. In *Proceedings of the 15th National Computer Security Conference*, pages 554–563. U.S. National Computer Security Center and U.S. National Bureau of Standards, 1992.

[12] Rebecca E. Field and Brant C. Jones. Using carry-truncated addition to analyze add-rotate-xor hash algorithms. *CoRR*, abs/1303.4448, 2013.

[13] Free Software Foundation. Gcc - dividing the output into sections. `https://gcc.gnu.org/onlinedocs/gccint/Sections.html`. Acessado em: 30 Mar. 2016.

[14] Michael Fox, John Giordano, Lori Stotler, and Arun Thomas. Selinux and grsecurity: A case study comparing linux security kernel enhancements, 2009.

[15] Michael Fox, John Giordano, Lori Stotler, and Arun Thomas. Selinux and grsecurity: Mandatory access control and access control list implementations., 2009.

[16] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *In Proc. Network and Distributed System Security Sym*, 2003.

[17] G. Scott Graham and Peter J. Denning. Protection: Principles and practice. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, AFIPS '72 (Spring), pages 417–429, New York, NY, USA, 1972. ACM.

[18] Bernhard Grill, Christian Platzer, and Jürgen Eckel. A practical approach for generic bootkit detection and prevention. In *Proceedings of the Seventh European Workshop on System Security*, EuroSec '14, pages 4:1–4:6, New York, NY, USA, 2014. ACM.

[19] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In Paul C. van Oorschot, editor, *USENIX Security Symposium*, pages 45–60. USENIX Association, 2008.

[20] Raphaël Hertzog and Roland Mas. O manual do administrador debian: Introducao ao selinux. `https://debian-handbook.info/browse/pt-BR/stable/sect.selinux.html`, 2015. Acessado em: 12 Jul. 2016.

[21] Vincent C. Hu, David Ferraiolo, Rick Kuhn, Arthur R. Friedman, Alan J. Lang, Margaret M. Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. Guide to attribute based access control (abac) definition and considerations (draft). Technical Report 800-162, NIST, 2013. Special Publication.

[22] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *IEEE Symposium on Security and Privacy*, pages 191–205. IEEE Computer Society, 2013.

[23] Haiku Inc. What is haiku? `https://www.haiku-os.org/about`, 2016. Acessado em: 12 Jul. 2016.

[24] Blexim Isen. Phrack inc. volume 0x0b. issue 0x3c. basic integer overflows. `http://www.phrack.com/issues/60/10.html#article`, 2002. Acessado em: 30 Mar. 2016.

[25] Herbert Xu James Morris, David S. Miller. Linux kernel cryptographic api. `http://lxr.free-electrons.com/source/Documentation/crypto/api-intro.txt`, 2015. Acessado em: 30 Mar. 2016.

[26] Aurélien Jarno. Debian squeeze and wheezy amd64 images for qemu or kvm. `https://people.debian.org/~aurel32/qemu/amd64/debian_wheezy_amd64_standard.qcow2`, 2014. Acessado em: 12 Jul. 2016.

[27] NIST John Kelsey. Sha-160: A truncation mode for sha256 and most other hashes. `http://csrc.nist.gov/groups/ST/hash/documents/Kelsey_Truncation.pdf`, 2005. Acessado em 22 Nov. 2016.

[28] Dave Jones. Trinity: Linux system call fuzzer. `https://github.com/kernelslacker/trinity`, 2015. Acessado em: 30 Mar. 2016.

[29] Anas Abou El Kalam, Salem Benferhat, Alexandre Miège, Rania El Baida, Frédéric Cuppens, Claire Saurel, Philippe Balbiani, Yves Deswarte, and Gilles Trouessin. Organization based access control. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, POLICY '03, pages 120–, Washington, DC, USA, 2003. IEEE Computer Society.

[30] Amit Kale, MontaVista Software Inc., and Wind River Systems Inc. Using kgdb, kdb and the kernel debugger internals. `https://www.kernel.org/pub/linux/kernel/people/jwessel/kdb/`, 2010. Acessado em: 26 Jul. 2016.

[31] Michael Kerrisk. ptrace(2) - linux man page. `http://linux.die.net/man/2/ptrace`, 2004. Acessado em: 30 Ago. 2013.

[32] Michael Kerrisk. Namespaces in operation, part 1: namespaces overview. `http://lwn.net/Articles/531114`, January 2013. Acessado em: 21 Jul.2013.

[33] Dustin Kirkland. Ecryptfs documentation. `http://ecryptfs.org/documentation.html`, 2012. Acessado em: 03 Abr. 2016.

[34] Sebastian Krahmer. Troubleshooter: The revenge of gingerbreak. `https://github.com/stealth/troubleshooter`, 2015. Acessado em: 26 Jul.2016.

[35] Cyberoam Technologies Pvt. Ltd. Identity and policy-based network security and management system and method, 03 2015.

[36] Andrea Mambretti, Kaan Onarlioglu, Collin Mulliner, William Robertson, Engin Kirda, Federico Maggi, and Stefano Zanero. Trellis: Privilege Separation for Multi-User Applications Made Easy. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, September 2016.

[37] Keegan McAllister. Attacking hardened linux systems with kernel jit spraying. `https://lwn.net/Articles/525609`, 2012. Acessado em: 30 Mar. 2016.

[38] Peter M. Mell and Timothy Grance. Sp 800-145. the nist definition of cloud computing. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2011.

[39] Victor Ramos Mello. Diamorphine: a lkm rootkit for linux kernels 2.6.x/3.x/4.x. `https://github.com/m0nad/Diamorphine`, 2015. Acessado em: 12 Jul. 2016.

[40] James Morris. Linux security module framework. `https://www.linux.com/learn/overview-linux-kernel-security-features`, 2013. Acessado em: 30 Mar. 2016.

[41] James Morris. Selinux project wiki. `http://selinuxproject.org`, 2013. Acessado em: 30 Mar. 2016.

[42] Binh Nguyen. Linux proc file system. `http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html`, 2004. Acessado em: 25 Jul.2016.

[43] Nick Nikiforakis, Frank Piessens, and Wouter Joosen. *HeapSentry: Kernel-Assisted Protection against Heap Overflows*, pages 177–196. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[44] Jon Oberheide and Dan Rosenberg. Stackjacking your way to grsec/pax bypass. https://jon.oberheide.org/files/stackjacking-infiltrate11.pdf, 2011.

[45] Kaan Onarlioglu, Collin Mulliner, William Robertson, and Engin Kirda. PrivExec: Private Execution as an Operating System Service. In *IEEE Symposium on Security and Privacy*, San Francisco, CA USA, May 2013.

[46] Kaan Onarlioglu, William Robertson, and Engin Kirda. Overhaul: Input-Driven Access Control for Better Privacy on Traditional Operating Systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2016.

[47] Bill Parducci. extensible access control markup language (xacml) specification, 2005.

[48] Christoph Lameter Paul Menage, Paul Jackson. Cgroups kernel documentation. `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`. Acessado em: 20 Jul.2016.

[49] Patrizio Pelliccione, Henry Muccini, Nicolas Guelfi, and Alexander Romanovsky. An introduction to software engineering and fault tolerance. *CoRR*, abs/1011.1551, 2010.

[50] Crispin Cowan Perry Wagle. Stackguard: Simple stack smash protection for gcc. Technical report, Immunix, Inc., 2003.

[51] P. A. H. Peterson. *Cryptkeeper: Improving security with encrypted RAM*. IEEE, Los Angeles, CA, USA, November 2010.

[52] Theofilos Petsios, Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. Dynaguard: Armoring canary-based protections against brute-force attacks. In *ACSAC*, pages 351–360. ACM, 2015.

[53] Phrack.0x0c41. Mistifying the debugger ultimate stealthness. `http://phrack.org/issues/65/8.html`, 2008. Acessado em: 30 Mar. 2016.

[54] Rob Pike, Dave Presotto, Sean Dorward Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 — the documents (volume 2). `http://plan9.bell-labs.com/sys/doc`, 2014. Acessado em: 21 Jul.2016.

[55] GNU project. Gdb: The gnu project debugger. `https://www.gnu.org/software/gdb/documentation/`, 2013. Acessado em: 06 Jun. 2016.

[56] RedHat. Systemtap. `https://sourceware.org/systemtap`, 2007. Acessado em: 30 Mar. 2016.

[57] RedHat. Systemtap script iotime. `https://sourceware.org/systemtap/examples/io/iotime.stp`, 2007. Acessado em: 30 Mar. 2016.

[58] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.

[59] Giovanni Russello, Arturo Blas Jimenez, Habib Naderi, and Wannes van der Mark. Firedroid: Hardening security in almost-stock android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 319–328, New York, NY, USA, 2013. ACM.

[60] Pierangela Samarati and Sabrina De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *Revised Versions of Lectures Given During the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design: Tutorial Lectures*, FOSAD '00, pages 137–196, London, UK, UK, 2001. Springer-Verlag.

[61] Casey Schaufler. Smack project description. `http://schaufler-ca.com/description_from_the_linux_source_tree`, 2011. Acessado em: 30 Mar. 2016.

[62] Thorsten Scherf. Grafische oberfläche für selinux. `http://www.admin-magazin.de/Das-Heft/2009/01/Grafische-Oberflaeche-fuer-SELinux`, 2009. Acessado em: 12 Jul. 2016.

[63] Z. Cliffe Schreuders, Tanya McGill, and Christian Payne. Empowering end users to confine their own applications: The results of a usability study comparing selinux, apparmor, and fbac-lsm. *ACM Trans. Inf. Syst. Secur.*, 14(2):19:1–19:28, September 2011.

[64] Z. Cliffe Schreuders, Christian Payne, and Tanya Mcgill. The functionality-based application confinement model. *Int. J. Inf. Secur.*, 12(5):393–422, October 2013.

[65] Scut. Exploiting format string vulnerabilities. Technical report, Team TESO, 2001.

[66] Adam Shanks. How-to su: Guidelines for problem-free su usage. `https://su.chainfire.eu`, 2015. Acessado em: 15 Jul. 2016.

[67] Otávio Augusto Silva, André Grégio, and Paulo Lício de Geus. Proteção de dados sensíveis através do isolamento de processos arbitrários em sistemas operacionais baseados no linu. In *XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, SBRC 2016*, Salvador, Bahia - Brasil, 2016.

[68] Rahul Simha, Alok Choudhary, Bhagi Narahari, and Joseph Zambreno. Security-driven compilation: A new direction for compiler research, 2004.

[69] Spender. Much ado about nothing: A response in text and code. `https://forums.grsecurity.net/viewtopic.php?f=7&t=2596`, 2011. Acessado em: 12 Jul. 2016.

[70] Spender. Grsecurity spender's exploit data base. `https://grsecurity.net/~spender/exploits/`, 2014. Acessado em: 12 Jul. 2016.

[71] Brad Spengler. Grsecurity comparison matrix. `https://grsecurity.net/compare.php`, 2015. Acessado em: 30 Mar. 2016.

[72] Brad Spengler. Grsecurity features. `https://grsecurity.net/features.php`, 2015. Acessado em: 30 Mar. 2016.

[73] Brad Spengler. Whats is grsecurity. `https://grsecurity.net/about`, 2015. Acessado em: 30 Mar. 2016.

[74] Dan Staples. Linux injector: Utility for injecting executable code into a running process on x86/x64 linux. `https://github.com/dismantl/linux-injector`, 2015. Acessado em: 06 Jun. 2016.

[75] Graham Steel. Deduction with XOR constraints in security API modelling. In Robert Nieuwenhuis, editor, *Proceedings of the 20th International Conference on Automated Deduction (CADE'05)*, volume 3632 of *Lecture Notes in Artificial Intelligence*, pages 322–336, Tallinn, Estonia, July 2005. Springer.

[76] National Security Agency Stephen Smalley. Flask: Flux advanced security kernel. `http://www.cs.utah.edu/flux/fluke/html/flask.html`, 2000. Acessado em: 25 Jul.2016.

[77] Daniel Svartman. Sudo until 1.8.14 - unauthorized privilege. `https://www.exploit-db.com/exploits/37710/`, 07 2015. Acessado em: 30 Mar. 2016.

[78] Andrew S. Tanenbaum. *Modern operating systems*, volume 3, chapter INTERPROCESS COMMUNICATION, page 145. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[79] Andrew S. Tanenbaum. *Modern operating systems*, volume 3, chapter MEMORY MANAGEMENT, page 248. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[80] Grsecurity Team. Grsecurity - the rbac system. `https://en.wikibooks.org/wiki/Grsecurity/The_RBAC_System`, 2013. Acessado em: 09 Jul. 2016.

[81] PaX Team. Pax - gcc plugins galore. `https://pax.grsecurity.net/docs/PaXTeam-H2HC13-PaX-gcc-plugins.pdf`, 2013. Acessado em: 06 Jun. 2016.

[82] PaX Team. Documentation for the pax project. `https://pax.grsecurity.net/docs`, 2015. Acessado em: 06 Jun. 2016.

[83] QubesOS Team. Qubes os project. `https://www.qubes-os.org`, 2015. Acessado em: 12 Jul. 2016.

[84] James Turnbull. Docker Container Breakout Proof-of-Concept Exploit. `https://blog.docker.com/2014/06/docker-container-breakout-proof-of-concept-exploit/`, 06 2014. Acessado em: 21 Jul.2016.

[85] US-CERT/NIST. Linux kernel 3.13.0 $<$ 3.19 (apparmor ubuntu $>=$ 15.04) - 'overlayfs' local root shell. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1328`, 2015. Acessado em: 21 Jul.2016.

[86] US-CERT/NIST. Multiple stack-based buffer overflows in the libresolv library in the gnu c library. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-7547`, 2015. Acessado em: 21 Jul.2016.

[87] US-CERT/NIST. Asus memory mapping driver (asmmap/asmmap64): Physical memory read/write. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0093`, 2016. Acessado em: 21 Jul.2016.

[88] US-CERT/NIST. Linux kernel before 4.4.1 mishandles object references in session keyring. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2016-0728`, 2016. Acessado em: 21 Jul.2016.

[89] Hayawardh Vijayakumar, Guruprasad Jakka, Sandra Rueda, Joshua Schiffman, and Trent Jaeger. Integrity walls: Finding attack surfaces from mandatory access control policies. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 75–76, New York, NY, USA, 2012. ACM.