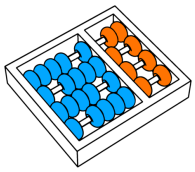
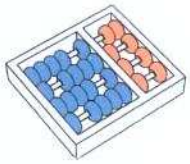


André Augusto da Silva Pereira

"Framework de kernel para Auto-proteção e Administração em um Sistema de Segurança Imunológico"

CAMPINAS
2013



Universidade Estadual de Campinas
Instituto de Computação

André Augusto da Silva Pereira

"Framework de kernel para Auto-proteção e Administração em um Sistema de Segurança Imunológico"

Orientador(a): **Prof. Dr. Paulo Lício de Geus**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

Este exemplar corresponde à versão final da Dissertação defendida por André Augusto da Silva Pereira, sob orientação de Prof. Dr. Paulo Lício de Geus.

Assinatura do Orientador(a)

CAMPINAS
2013

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

P414f

Pereira, André Augusto da Silva, 1986-
Framework de kernel para auto-proteção e administração em um sistema de segurança imunológico / André Augusto da Silva Pereira. – Campinas, SP : [s.n.], 2013.

Orientador: Paulo Lício de Geus.
Dissertação \\\(mestrado\\\) – Universidade Estadual de Campinas, Instituto de Computação.

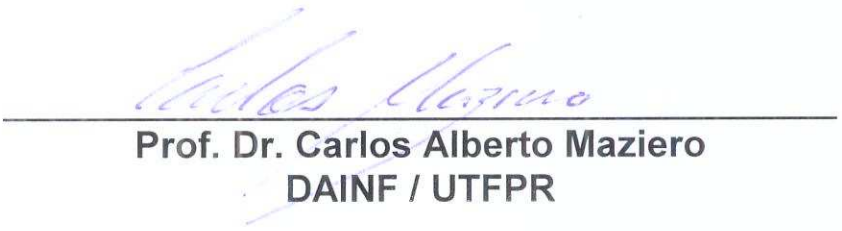
1. Sistemas operacionais (Computadores). 2. Redes de computadores - Sistemas de segurança. 3. Imunoinformática. I. Geus, Paulo Lício de,1956-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em inglês: A kernel framework for administration and selfprotection for a immunological security system
Palavras-chave em inglês:
Operating systems (Computers)
Computer networks - Security measures
Immunoinformatics
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Paulo Lício de Geus [Orientador]
Ricardo Dahab
Carlos Alberto Maziero
Data de defesa: 19-04-2013
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 19 de Abril de 2013, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Carlos Alberto Maziero
DAINF / UTFPR

Prof. Dr. Ricardo Dahab
IC / UNICAMP

Prof. Dr. Paulo Lício de Geus
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Framework de kernel para Auto-proteção e Administração em um Sistema de Segurança Imunológico

André Augusto da Silva Pereira[1]

19 de abril de 2013

Banca Examinadora:

- Prof. Dr. Paulo Lício de Geus
  Instituto de Computação, UNICAMP (Orientador)
- Prof. Dr. Ricardo Dahab
  Instituto de Computação, UNICAMP
- Prof. Dr. Carlos Alberto Maziero
  Departamento Acadêmico de Informática, UTFPR
- Prof. Dr. Julio Cesar López Hernández
  Instituto de Computação, UNICAMP (suplente)
- Prof. Dr. Adriano Mauro Cansian
  Instituto de Biociências Letras e Ciências Exatas, UNESP (suplente)

[1]Suporte financeiro de: Bolsa CAPES 2009 - 2010, Bolsa FAPESP (processo 2009/13240-4) 2010 - 2011

Abstract

According to the traditional view in computer system security, the correct specification and implementation of security policies, along with the correct implementation and proper configuration of computer systems, are enough to guarantee that a system is secure [33]. It happens, though, that both policy and system implementations are subject to failures that would lead to a compromise of the system's security as a whole.

By seeking for inspiration on nature's systems, one can see that the human immune system provides a model that is very close to how computer systems normally have to behave for their security. That way, seeking for relations between the computational and the biological models provides a rich source for research inspiration.

This work reviews the research made at the Criptography and Security Laboratory (LASCA), University of Campinas, Brazil, in order to develop a fully functional immunological security system. It also implements, in a simple to use and easy to maintain manner, a Linux kernel framework aimed to provide all the requirements needed by this research effort in order to build such system.

The developed framework is a Linux Security Module (LSM) and makes use of a set of consolidated Linux tools in order to provide such requirements, like Task Control Group (TCG) and Netfilter.

Keywords: *Intrusion Detection System, Computer Immunology, Immune System, Operating Systems, Linux kernel, Resource Control*

Resumo

Segundo a visão tradicional de segurança de sistemas computacionais, a especificação e implantação de políticas de segurança em conjunto com implementações corretas e configuração adequada garantem a segurança de um sistema [33]. Na prática, contudo, percebe-se que tanto políticas quanto implementações estão sujeitas a falhas que comprometem a segurança do sistema como um todo.

Buscando inspiração na natureza, o sistema imunológico humano possui características que representam um modelo bastante próximo das condições em que sistemas computacionais se encontram. Desta forma, a exploração de paralelos com este modelo biológico é uma rica fonte de inspiração para a pesquisa em segurança de sistemas computacionais.

Esta dissertação apresenta os trabalhos realizados por pesquisadores do Laboratório de Segurança e Criptografia, na busca pela construção de um sistema de segurança imunológica geral, e implementa um *framework* no nível de *kernel* em Linux, com objetivo de prover todos os requisitos necessários à implementação do sistema proposto por esses pesquisadores, através de um sistema com implementação clara, funcional, de fácil manutenção e API simplificada.

O *framework* desenvolvido é implementado como um *Linux Security Module* e agrega algumas ferramentas consolidadas do *kernel* Linux em sua API, tais como os *frameworks Task Control Group* (TCG) e *Netfilter*.

Palavras-chave: Sistema de Detecção de Intrusão, Imunologia Computacional, Sistema Imunológico, Sistemas operacionais, Segurança de sistemas operacionais, *kernel* Linux, Controle de recursos.

Dedico este trabalho à minha esposa, Tamyres, e à minha filha, Ana Clara. Sem vocês a vida não teria sentido.

Agradecimentos

Eu gostaria de agradecer à minha mãe, Edna, que desde cedo, com seus sábios conselhos ao pé do ouvido, moldou o meu caráter e os meus valores. Ao meu pai, Carlos, que com toda a experiência acumulada ao longo de sua difícil, porém vitoriosa, trajetória de vida, esteve sempre vigilante, orientando-me e preparando-me para a vida. À Deus e aos meus pais, credito todos os meus passos até este ponto.

Gostaria de agradecer à minha esposa, Tamyres Pereira, por assumir, durante os meus períodos de ausência, os papéis de pai e mãe da nossa querida Ana Clara, suportando todas as dificuldades, e me apoiando incondicionalmente nesta jornada. Além de todo esse apoio, a sua confiança em mim foi fundamental nos momentos de dúvida. Sem você eu não teria conseguido. Te amo.

Agradeço aos colegas do Laboratório de Administração e Sistemas de Segurança, Victor Ammaduci, Arthur Castro, Edmar Azevedo, Gabriel Cavalcante, Ricardo Saffi, Cléber Paiva e Miguel Gaiowski, por toda a hospitalidade e ajuda no desenvolvimento deste trabalho. Agradeço também ao meu orientador, Prof. Dr. Paulo Lício de Geus, por ter me aceitado como seu aluno e compartilhado do seu conhecimento.

Gostaria de prestar meus agradecimentos ainda ao meu colega, Carlos Macapuna, que além de toda a ajuda durante a nossa graduação, mudou-se para Campinas antes de minha chegada e ajudou bastante na minha transição de Belém para Campinas.

Finalmente, agradeço aos órgãos de fomento à pesquisa, CAPES e FAPESP, pelo apoio financeiro.

"Somos o que repetidamente fazemos.
A excelência, portanto, não é um feito,
mas um hábito."
Aristóteles

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Segundo a visão tradicional de segurança de sistemas computacionais, a especificação e implantação de políticas de segurança em conjunto com implementações corretas e configuração adequada garantem a segurança de um sistema [33]. Na prática, contudo, percebe-se que tanto políticas quanto implementações estão sujeitas a falhas que comprometem a segurança do sistema como um todo.

Neste contexto, necessita-se de novos mecanismos que garantam a segurança de sistemas mesmo na existência dessas eventuais falhas. Deseja-se, portanto, um novo nível de segurança, com recursos adicionais e modelos que representem melhor as condições em que os sistemas computacionais encontram-se.

Partindo desta necessidade, diversos pesquisadores buscaram inspiração em modelos existentes na natureza, e encontraram um que representa muito bem o problema de segurança em sistemas computacionais: o Sistema Imunológico Humano.

Além de representar um modelo bastante próximo das condições em que a maioria das redes de computadores se encontra, o sistema imunológico humano possui uma série de características desejáveis a um sistema de segurança, tais como detecção de anomalias, elaboração de plano de resposta e contra-ataque [23].

Em 1999, o Laboratório de Segurança e Criptografia (LASCA) da Universidade Estadual de Campinas (UNICAMP), iniciou um projeto de pesquisa chamado Imuno, o qual se concentra no estudo dos princípios do sistema imunológico humano aplicados em segurança computacional.

O foco daquele projeto era de explorar os paralelos entre o sistema imunológico humano e sistemas de segurança computacional dentro de uma visão integrada, e não particularizada em objetivos específicos, tal como detecção de intrução ou recuperação de ataques. O projeto Imuno, portanto, busca desde então construir um sistema de segurança imunológica geral, capaz de enfrentar ameaças desde o seu primeiro contato, passando por medidas de contingência e resposta, até a total recuperação do sistema, tal como o modelo

biológico.

Como fruto desta pesquisa, diversos protótipos de segurança imunológica foram desenvolvidos [22] [20] [30]. Estes protótipos têm em comum uma implementação complexa, de difícil manutenção, e com grande utilização de recursos de núcleo do sistema operacional onde são implementados.

Com o crescimento da base de código da solução proposta inicialmente, a complexidade inerente à implementação do sistema de segurança imunológico geral almejado pelo projeto passou a representar uma barreira para o avanço da pesquisa. Neste contexto, em 2006, Martim Carbone propôs um *framework* a ser utilizado como base comum para implementação dos diversos componentes do sistema de segurança em desenvolvimento. O objetivo da ferramenta era reunir toda a complexidade de implementação de recursos a nível de núcleo de sistema operacional, provendo uma API simplificada, em espaço de usuário, aos demais pesquisadores do projeto Imuno, que poderiam, então, concentrar ainda mais seus esforços na pesquisa de soluções imunológicas, diminuindo a grande carga que a implementação de recursos no nível de *kernel* representava.

O trabalho de Carbone obteve sucesso na comprovação de sua tese, resultando em um protótipo capaz de comprovar a viabilidade de sua solução.

Este trabalho, portanto, vem dar continuidade àquele, e tem como objetivo entregar um *framework* funcional aos pesquisadores do projeto Imuno, através da consolidação e extensão do *framework* imunológico proposto por Carbone.

Este texto é organizado da seguinte forma: o Capítulo 2 inicia a revisão bibliográfica realizada nesta pesquisa, apresentando alguns conceitos fundamentais sobre o funcionamento do sistema imunológico humano e os objetivos das pesquisas na área de segurança imunológica. Neste capítulo serão apresentados também todos os trabalhos desenvolvidos pelo projeto Imuno até o presente momento, além de referências a pesquisas relacionadas.

Avançando com a revisão bibliográfica, o Capítulo 3 apresentá o funcionamento do *kernel* Linux, plataforma sobre a qual este trabalho foi implementado, discutindo algumas de suas características fundamentais e introduzindo importantes estruturas do sistema que serão amplamente utilizadas neste trabalho.

Finalizando a revisão bibliográfica, o Capítulo 4 traz uma visão de diversos aspectos de segurança do *kernel* Linux, apresentando algumas de suas importantes ferramentas.

O Capítulo 5 discute o trabalho realizado nesta pesquisa. Neste capítulo serão discutidas todas as etapas de desenvolvimento do *framework* IMN, desde sua concepção, através de sua análise de requisitos, passando pelos aspectos de projeto e implementação da solução. Por fim, no Capítulo 6, será feita uma análise do trabalho desenvolvido, bem como alguns testes funcionais, a partir do desenvolvimento de um sistema de segurança simplificado, capaz de exercitar as diferentes funcionalidades disponibilizadas pela API do *framework* desenvolvido.

O Capítulo 7 encerra esta dissertação com algumas considerações finais sobre o trabalho e as principais linhas de pesquisa futura a serem seguidas pelo autor.

Capítulo 2

Imunologia

2.1 Imunologia Biológica

O Sistema Imunológico Humano é um sistema biológico composto de diversas células, órgãos e tecidos, cuja principal função é proteger o corpo contra a invasão de micróbios[1], denominados *patógenos* [52].

A chave para o funcionamento do sistema imunológico está na sua habilidade de diferenciar células *self* (ou seja, que fazem parte do corpo) de células *non-self* (estranhas ao corpo) e na sua capacidade de eliminar as células *non-self*.

O sistema imunológico humano é dividido em sistema inato e sistema adaptativo. O sistema imunológico inato representa a primeira linha de defesa do organismo e possui resposta não-específica (a mesma para a maioria dos patógenos), sendo muitas vezes insuficiente. Seus principais componentes são as barreiras físicas e químicas (pele, ácidos gástricos, proteínas do sangue) e células de detecção não-específica (fagócitos). O sistema imunológico inato tem origem congênita e hereditária, permanecendo inalterado ao longo da vida.

Já o sistema adaptativo é capaz de identificar especializadamente um agente patogênico, respodendo de maneira mais eficiente. Este sistema tem capacidade de aprendizado e auto-ajuste, sendo capaz de memorizar um agente e responder de forma mais rápida e eficiente no caso de uma nova exposição.

O sistema imunológico adaptativo é a segunda camada de defesa do sistema imunológico e seus componentes básicos são células denominadas linfócitos, que podem ser dos tipos B e T, e seus produtos: anticorpos e linfocinas. Ao contrário do sistema inato, onde um receptor pode reconhecer vários antígenos diferentes, no sistema adaptativo os receptores dos linfócitos realizam reconhecimento específico.

[1]Micróbios são pequenos organismos causadores de infecções, tais como bactérias, vírus, parasitas e fungos.

O funcionamento geral do sistema imunológico é como segue: primeiro os patógenos devem vencer as barreiras físicas e químicas do sistema inato, ou seja, penetrar na pele ou sobreviver aos ácidos estomacais. Caso essa primeira etapa seja vencida e o micróbio adentre o organismo, é hora do sistema imunológico iniciar a sua resposta, a qual é dividida nas fases de detecção, ativação do sistema adaptativo e contra-ataque [23].

A **fase de detecção** ocorre quando os receptores dos fagócitos reagem com os antígenos dos patógenos na corrente sanguínea (Figura 2.1). Nestes casos, inicia-se a fagocitose, processo pelo qual uma célula envolve uma partícula com seu próprio corpo, dissolvendo o micróbio em seu interior. Com essa dissolução, o fagócito passa a produzir moléculas de proteína chamadas MHC (*Major Histocompatibility Complex*) contendo fragmentos do antígeno do micróbio. Estas moléculas são expelidas pela membrana plasmática dos fagócitos.

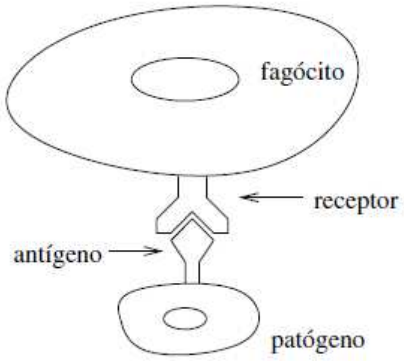


Figura 2.1: Reação entre receptor e antígeno.

A **fase de ativação do sistema adaptativo** inicia-se com um processo de apresentação dos antígenos estranhos aos linfócitos T, capazes de reconhecer antígenos específicos nas moléculas MHC na superfície dos fagócitos. Após esta ativação química do sistema adaptativo, os linfócitos B e T atraídos para o local reproduzem-se[2], aumentando o exército de combate. Em seguida, estes linfócitos passam por um processo de *maturação de afinidade*, no qual são evoluídos de forma a aumentar ainda mais a sua afinidade com o antígeno em questão. Parte destes linfócitos são convertidos em células de memória e passam a constituir uma memória imunológica, a qual contém informações sobre os antígenos que serão usadas em futuras exposições.

Em seguida, inicia-se a **fase de contra-ataque** (também conhecida como resposta secundária ou específica). Estimulados pelos linfócitos T, os linfócitos B iniciam a produção de anticorpos, que então reagem com os antígenos presentes na superfície dos patógenos,

[2]Processo conhecido como evolução clonal

deixando-os marcados para destruição por parte dos fagócitos. Os linfócitos T também destroem as células infectadas, a fim de impedir a reprodução do organismo invasor.

Finalmente a infecção é contida e os estímulos imunológicos vão diminuindo até que a resposta imunológica chegue ao fim.

O processo descrito deixa clara a existência de diversas camadas de proteção existentes no sistema imunológico. A Figura 2.2 ilustra estas diferentes camadas de proteção.

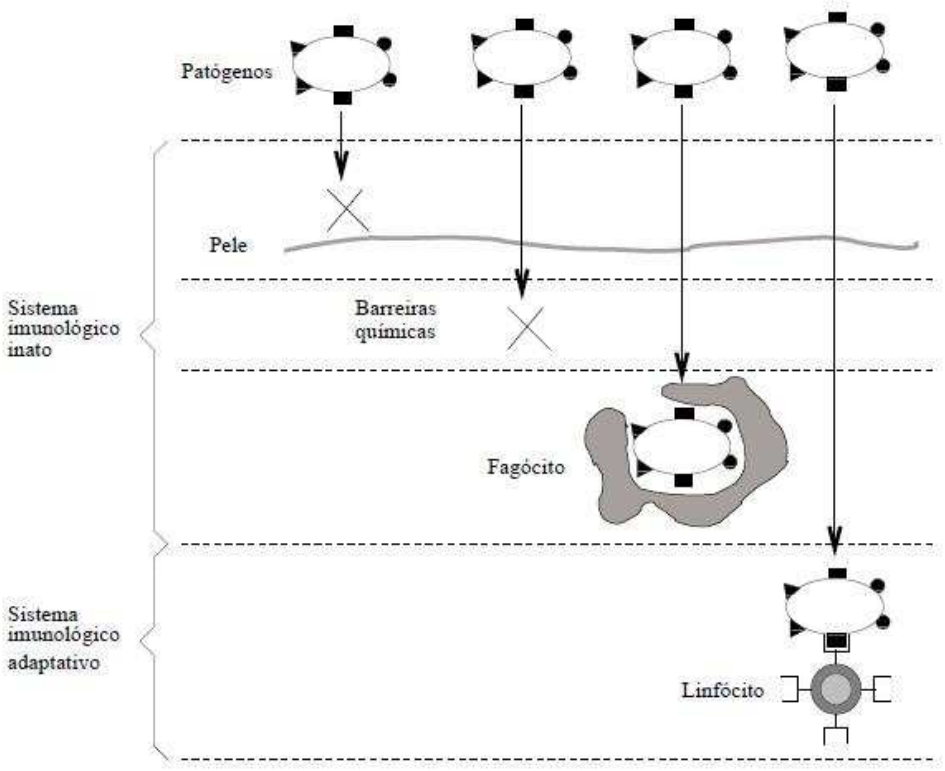


Figura 2.2: Camadas de proteção do sistema imunológico humano.

O sistema imunológico, portanto, possui uma série de características que conferem eficácia na tarefa de manutenção da vida humana, através da proteção do organismo contra ataques de agentes externos. Estas características são:

- Distribuição e paralelismo: a detecção de uma célula de defesa não é iniciada por algum mecanismo centralizador, mas quando alguma condição não usual é detectada pela célula. Ademais, o sistema imunológico pode atuar em diferentes pontos do organismo em um dado momento;

- Multiplas camadas de proteção: a atuação do sistema imunológico humano acontece através de diferentes mecanismos especializados que, cooperativamente, garantem a

proteção do organismo;

- Diversidade: introduzindo uma variedade de indivíduos com características particulares de defesa é possível garantir a sobrevivência de uma espécie;
- Descartabilidade: nenhum componente isolado do sistema imunológico é essencial;
- Autonomia: o sistema imunológico como um todo não requer gerenciamento ou manutenção externos;
- Adaptabilidade e memória: o sistema imunológico possui a capacidade de assimilar a detecção de novos patógenos, armazenando a habilidade de reconhecimento adquirida na memória imunológica. A capacidade de reconhecimento especializa-se a cada agente patogênico similar reconhecido, tornando as resposta futuras mais eficientes se comparadas às anteriores;
- Mudança dinâmica de cobertura: visando maximizar a capacidade de detecção, o sistema imunológico mantém uma amostra aleatória de detectores que circulam pelo corpo. Essa amostra é constantemente renovada através da morte de células e a produção de novas células;
- Identificação por comportamento: uma célula pode ser analisada através das substâncias que ela secreta;
- Detecção de anomalia: o sistema imunológico tem a capacidade de detectar agentes patogênicos através da identificação de atividade não usual;
- Detecção de danos: muitos patógenos não causam ameaça ao corpo e, nesse caso, uma resposta imunológica para eliminá-los poderia prejudicar o próprio corpo. Entretanto, a ocorrência de danos em tecidos de um organismo é essencial para a ativação dos linfócitos T, que irão sobreviver equanto houver evidências desses danos. Essa característica coestimulatória diminui as chances do sistema imunológico desencadear uma reação autoimune;
- Detecção imperfeita: por possibilitar uma detecção imperfeita ou parcial, o sistema imunológico aumenta a flexibilidade na alocação de recursos. Desta forma, detectores menos específicos podem responder a uma variedade maior de patógenos, contudo irão ser menos eficientes na detecção de um patógeno específico;
- Recrutamento dinâmico: o sistema imunológico regula o número de células de maneira a desenvolver um contra-ataque suficiente para a quantidade atual de agentes patogênicos. Assim que a ameaça é eliminada, essa quantidade de células é reduzida.

2.2 Imunologia Computacional

Diversos sistemas biológicos costumam ser estudados para fornecer inspiração para o desenvolvimento de soluções da ciência e engenharia. Dentre eles, o sistema imunológico humano possui uma série de princípios que podem ser aplicados em segurança computacional. A Imunologia Computacional explora estes princípios com o objetivo de reproduzir o comportamento do sistema imunológico humano em ambientes computacionais, visando uma elevação dos níveis de segurança destes ambientes.

Em 1999, no Laboratório de Segurança e Criptografia (LASCA) da Universidade Estadual de Campinas (UNICAMP), iniciou-se um projeto de pesquisa denominado Imuno. Os pesquisadores deste projeto revisaram trabalhos de pesquisa em imunologia computacional e, motivados pelas diversas analogias observadas entre o sistema imunológico humano e as necessidades de um sistema de segurança computacional, propuseram o desenvolvimento de uma arquitetura de segurança baseada no modelo do sistema imunológico humano[23].

Após estudo criterioso do funcionamento do sistema imunológico humano, o grupo constatou que as pesquisas que vinham sendo desenvolvidas naquela época focavam apenas em mecanismos isolados do sistema imunológico, deixando de lado a visão do seu funcionamento integrado. Por este motivo, o projeto Imuno trabalha desde então em pesquisas relacionada à construção de um sistema de segurança imunológico geral, onde seja possível detectar-se anomalias, elaborar um plano de resposta especializado e efetuar o contra-ataque. O sistema deve ainda ter capacidade de aprendizado e adaptação, possibilitando, portanto, reação contra ataques desconhecidos.

A proposta original [23] foi refinada [24] [25] e os principais aspectos de segurança imunológica, tais como análise forense [30] e resposta automatizada [20] foram estudados, culminando com a concepção de uma Arquitetura para Sistema de Segurança Imunológico Computacional completo[22] [25] [24].

Em seguida, foi feita a implementação de um protótipo chamado ADenoIDS (*Acquired Defense System based on the Immune System*), o qual implementa parcialmente a arquitetura proposta no sistema operacional GNU/Linux. O protótipo ADenoIDS é concentrado nas funcionalidades de detecção e resposta automática para ataques do tipo *buffer overflow* remotos, podendo ser visto como um protótipo do sistema Imuno com escopo reduzido.

Com o objetivo de facilitar o processo de desenvolvimento do sistema de segurança imunológico completo inicialmente proposto, em 2006 Carbone [29] iniciou o desenvolvimento de um *framework*[3] de segurança imunológica em ambiente Linux[4].

[3]Um *framework* é uma abstração que une códigos comuns entre vários projetos de *software*, provendo uma funcionalidade genérica

[4]Neste documento o termo "Linux" será utilizado para referenciar o núcleo (*kernel*) do sistema operacional GNU/Linux, desenvolvido por Linus Torvalds.

Estes trabalhos são apresentados com mais detalhes a seguir, nas Seções 2.2.1, 2.2.2, 2.2.3, 2.2.4 e 2.2.5. Na Seção 2.2.6, serão listados outros projetos de pesquisa relacionados à imunologia computacional.

2.2.1 Modelagem de um sistema de segurança imunológico

Neste trabalho, Diego Fernandes, Fabrício de Paula e Marcelo Reis estabeleceram uma analogia entre imunologia e segurança de sistemas computacionais (Tabela 2.1).

Sistema Imunológico	**Segurança Computacional**
Sistema de filtragem composto por cílios, pele, mucosas e ácidos	*Firewall* de rede
Amígdalas	Bode expiatório (*boddy traps*) armados pelo administrador de segurança
Inserção de DNA de vírus no interior de células atacadas	*Buffer overflow* e instalação de *trojan horses*
Detecção de agentes patogênicos por macrófagos	Sistema de detecção de intrusão
Produção de moléculas MHC com fragmentos de antígenos	Análise do comportamento de processos
Ativação de linfócitos específicos	Análise forense
Antígenos de agentes patogênicos	Assinaturas digitais de ataques
Memória Imunológica	Alimentação de uma base de dados
Destruição de células contaminadas e de agentes patogênicos	Medidas de contenção, tais como finalização de processos e encerramento de conexões

Tabela 2.1: Paralelos entre o sistema imunológico e segurança de redes de computadores [23].

Avançando na análise dos paralelos biológicos computacionais, alguns príncipios organizacionais do sistema imunológico foram considerados base importante para o desenvolvimento de um sistema de segurança computacional:

- Descentralização e localidade: a ação de uma célula de defesa não é iniciada por um mecanismo centralizador, mas sim pela própria célula na constatação de alguma anormalidade. Esta abordagem elimina a existência de um ponto central de falha e permite a atuação paralela em diferentes pontos de infecção;

- Arquitetura multi-camada: o funcionamento do sistema imunológico depende da combinação de vários mecanismos diferentes de desesa. Cada um destes mecanismos é especializado e resolve bem pequenos problemas. A combinação destes mecanismos é o que garante a defesa completa do organismo;

- Diversidade: o corpo humano pode ser exposto a um grande número de agentes patogênicos. Por este motivo, o sistema imunológico é capaz de identificar uma grande variedade de invasores e agir de maneira especializada para cada um deles;

- Robustez e tolerância a falhas: nenhuma célula componente de algum mecanismo é essencial, por isso, em caso de comprometimento (infecção ou morte) de uma célula, não há comprometimento do funcionamento do sistema;

- Autonomia: o sistema imunológico humano não requer nenhum tipo de gerenciamento externo para promover o seu funcionamento;

- Adaptabilidade e memória: apesar de ser gerado (nascer) com algumas defesas prontas, o sistema imunológico é capaz também de identificar agentes patogênicos totalmente desconhecidos e desenvolver uma estratégia de defesa contra eles. Uma vez que esta habilidade é desenvolvida, o sistema a perpetua, construindo uma memória biológica, de tal forma que em caso de nova exposição a estratégia de defesa já seja conhecida;

- Auto-proteção: o sistema imunológico é capaz de defender não somente células de outros sistemas humanos, mas também é capaz de proteger as suas próprias células, uma vez que estas também podem ser vítimas de ataque de agentes patogênicos;

- Identificação por meio de comportamento: além da capacidade de detectar agentes patogênicos, o sistema imunológico é capaz também de detectar anomalias através do comportamento de células próprias do organismo, concluindo que estão contaminadas e atacando-as;

- Tamanho do exército: o sistema imunológico humano é capaz de regular o tamanho do ataque a agentes patogênicos, em termos de número de linfócitos utilizados, de maneira proporcional e suficiente.

Por fim o trabalho estabeleceu um modelo estrutural e um modelo de funcionamento de um sistema computacional de segurança imunológico, composto pelos sistemas inato e adaptativo. O modelo estrutural definiu componentes capazes atender aos princípios organizacionais do sistema imunológico anteriormente apresentados. As interações entre os componentes do modelo estrutural devem gerar comportamento global análogo ao sistema imunológico humano. O modelo estrutural proposto é apresentado na Figura 2.3. O funcionamento deste modelo é ilustrado na Figura 2.4.

2.2.2 Forense computacional aplicada em segurança imunológica

Forense computacional é uma forma de detecção de intrusão póstuma baseada em evidências, ou seja, uma invasão somente é detectada após ter ocorrido com sucesso,

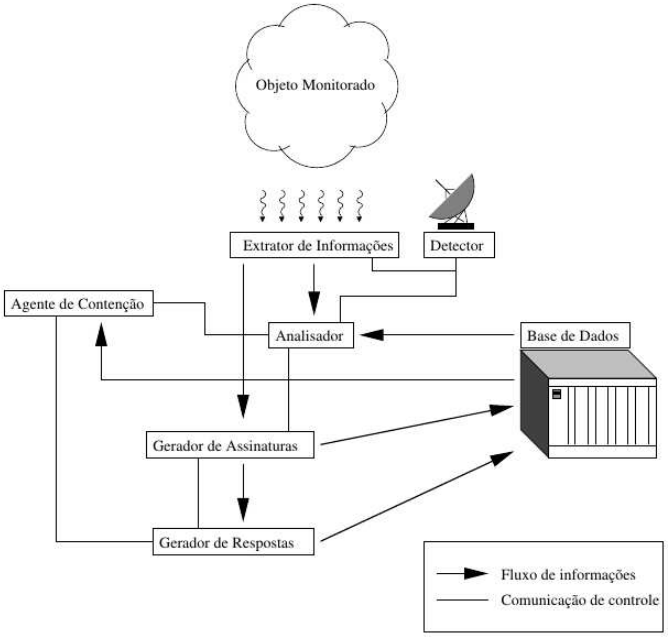


Figura 2.3: Modelo Estrutural proposto em [23].

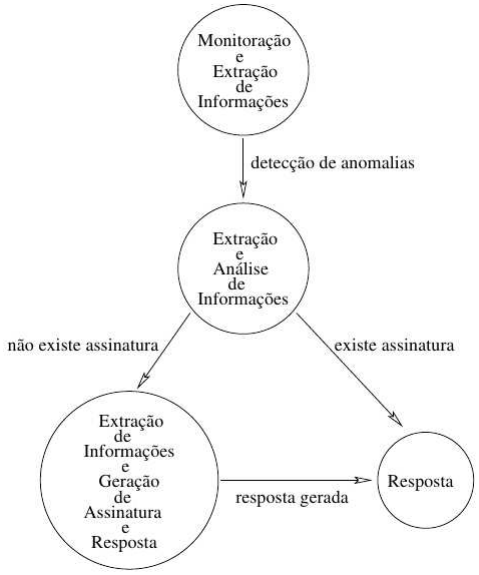


Figura 2.4: Modelo de funcionamento proposto em [23].

através da análise feita sobre o estado e o comportamento (evidências) do sistema comprometido.

Do ponto de vista imunológico, um analisador forense automatizado corresponderia ao módulo detector, conforme no modelo estrutural de sistema imunológico apresentado na Figura 2.3.

A análise forense correlaciona diversas evidências para definir a assinatura de um ataque. Dentre estas evidências podem estar o tipo de falha explorada, as ações do invasor, origem do ataque e outros. Estas evidências e suas inter-relações podem ser descritas previamente, baseando-se na experiência do investigador. Desta forma, pode-se criar uma base de dados de evidências forenses e utilizá-la em um sistema automatizado de análise forense.

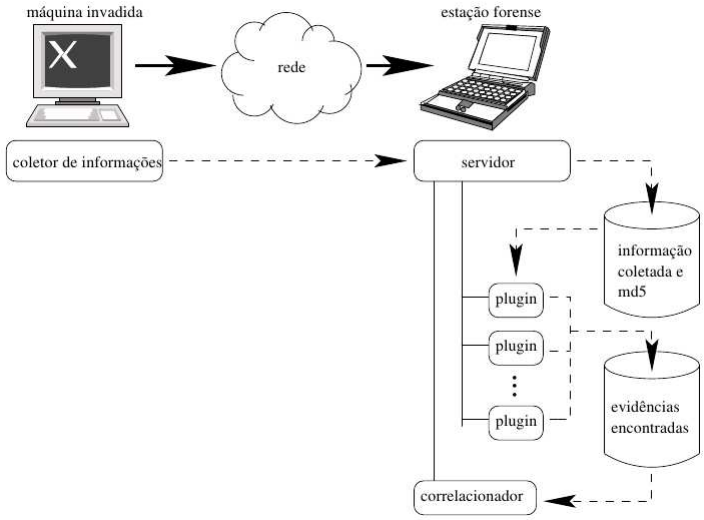


Figura 2.5: Arquitetura de um analisador forense automatizado proposta em [30].

Macelo Reis, em sua tese de mestrado, criou uma arquitetura extensível para desenvolvimento de um analisador forense automatizado capaz de coletar informações, identificar e correlacionar evidências de uma intrusão. A arquitetura proposta utiliza um modelo distribuido do tipo cliente-servidor, onde o cliente possui um módulo coletor de informações e o servidor, batizado de estação forense, tem o papel de identificar e correlacionar as evidências recebidas do cliente, na tentativa de detectar eventuais ataques naquela máquina. A Figura 2.5 ilustra esta arquitetura.

O investigador deve determinar quais informações devem ser coletadas pelo módulo coletor. Uma vez coletados, os dados devem ser enviados para o servidor utilizando preferencialmente um canal seguro. O módulo coletor reune informações de fontes de possíveis evidências, abaixo listadas:

- registradores e cache
- memória
- estado do sistema (conexões de rede, tabela de roteamento, processos em execução, usuários logados, data e hora do sistema, dentre outras)
- tráfego de rede
- dispositivos de armazenagem secundária (análise de dados em disco)

A busca e extração de evidências é feita no servidor, de forma automatizada, através da utilização de *plugins* especializados em subconjuntos ou categorias de evidências. Segundo o autor, diferentes técnicas de detecção de intrução baseadas em evidências estão disponíveis, cada uma com pontos fortes e fracos. Contudo, nenhuma delas é totalmente eficaz [7]. Desta forma, cada *plugin* representa uma técnica de detecção, e o empilhamento destes oferece um aumento da eficácia global do sistema de detecção, conferindo a característica de extensibilidade à arquitetura.

O autor iniciou um protópipo, o AFA (*Automated Forensic Analyser*). Este protótipo implementou a arquitetura proposta limitando-se à coleta de informações na máquina cliente e, no servidor, desenvolvimento de um *plugin*. Não houve, contudo, testes para exercitar o protótipo.

2.2.3 Resposta Automática em Segurança Imunológica

Dentro da visão de construção de um sistema de segurança imunológico geral, Diego Fernandes, em sua dissertação de mestrado, trabalhou no desenvolvimento de um **mecanismo de resposta automática** [20], parte integrante de um sistema de segurança imunológico, representando o módulo "Agente de Contenção" do modelo apresentado na Figura 2.3.

Mecanismos de resposta devem disponibilizar um conjunto de recursos capazes de retardar a ação de um ataque, bloqueá-lo e restaurar o sistema a um estado seguro. Ataques conhecidos devem possuir mecanismos específicos e resposta, ao passo que devem existir esquemas de contenção para ataques desconhecidos, visando reduzir o seu impacto, uma vez que não existem soluções específicas estes ataques.

O mecanismo de resposta desenvolvido é dividido em dois esquemas de resposta: primária e secundária. As ações de resposta do mecanismo são classificadas em níveis. Ações em nível de processo atuam no sentido de limitar a capacidade de processamento de processos suspeitos, bem como restringir o acesso destes a outros recursos de sistema. Já as ações em nível de rede atuam sobre o tráfego de rede gerado por estes processos

suspeitos. As ações desenvolvidas dispõem da capacidade de revogação, para que em casos onde processos sob suspeita sejam considerados legítimos, possa-se retornar à sua execução normal. Ambos esquemas de resposta, primário e secundários, atuam nestes dois níveis.

O mecanismo possui uma arquitetura distribuida e utiliza tecnologia de agentes móveis. A Figura 2.6 ilustra a arquitetura do mecanismo, onde cada máquina da rede (ponto de resposta) em um dado instante pode ter diferentes agentes móveis. Estes agentes, abaixo listados, possuem funcionalidades distintas e interagem entre si de forma a coordenar os esquemas de respostas.

- `initAgent`: agente de inicialização. Responsável pela inicialização do sistema de resposta;
- `primaryresponseAgent`: agente de resposta primária. Responsável pelo gerenciamento de respostas primárias em um ponto de resposta;
- `secondaryresponseAgent`: agente de resposta secundária. Responsável pelo gerenciamento de respostas secundárias em um ponto de resposta;
- `alarmAgent`: agente de alarme. Responsável pela comunicação entre os pontos de resposta;
- `logAgent`: agente de registro. Responsável por registrar em disco os eventos ocorridos no mecanismo de resposta;
- `directReactionAgent`: agente de reação direta. Responsável pelo gerenciamento de ações de conteção diretas em um ponto de resposta;
- `stimulatedReactionAgent`: agente de reação estimulada. Responsável pelo gerenciamento de ações de contenção estimuladas em um ponto de resposta.

Os agentes de resposta fazem uso de componentes locais de reação, os quais dão suporte aos agentes de resposta, implementando e executando todas as ações possíveis de uma resposta. O mecanismo implementado considera a diponibilidade das seguintes ações em componente local de reação:

- Nível de processos:
  - Limite de uso de memória: capacidade de limitar a quantidade de memória física utilizada por um processo;
  - Limite de consumo de CPU: capacidade de limitar o tempo de CPU utilizado por um processo;

  - Número máximo de conexões: capacidade de limitar o número de conexões abertas por um processo;
  - Quantidade de processos filhos: capacidade de limitar o número de processos filhos criados por um processo;
  - Limite de transações de acesso a disco: capacidade de limitar a leitura ou escrita de dados em disco, em termos de volume de dados, por um processo;
  - Limite de acesso a disco: capacidade de limitar operações de leitura ou escrita em disco, em termos de permissões, por um processo;
  - Finalização de processos: capacidade de finalizar processos;
  - Paralização de processos: capacidade paralização e eventual continuação na execução de processos.

- Nível de rede:
  - Bloqueio de tráfego: capacidade de bloquear tráfego por máquina, serviço, protocolo ou porta;
  - Limite de conexões: capacidade de limitar o número de conexões por serviço ou porta;
  - Bloqueio de tráfego por usuário: capacidade de bloquear tráfego de rede por usuário.

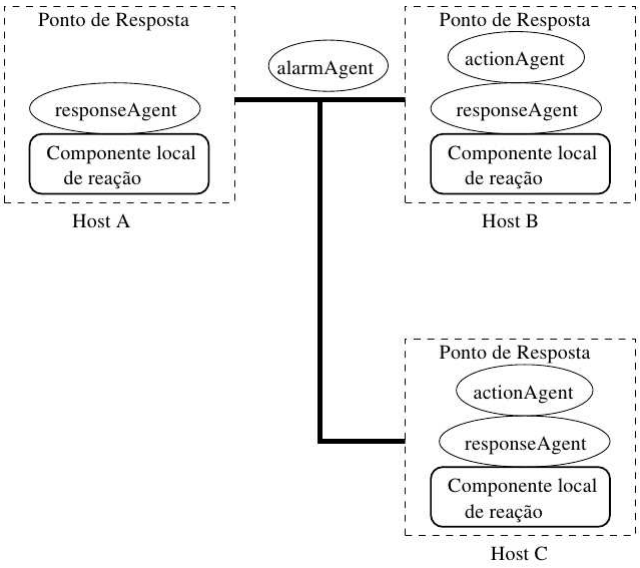


Figura 2.6: Mecanismo de resposta distribuído usando agentes móveis [20].

O funcionamento do mecanismo possui uma etapa de inicialização, onde um `initAgent` é criado. Este agente, por sua vez, cria agentes de resposta primária e secundária (`primaryresponseAgent` e `secondaryresponseAgent`) para cada ponto de resposta. Uma vez criados, os agentes de resposta movimentam-se para os seus respectivos pontos de resposta e criam um agente de registro (`logAgent`) para cada ponto. Uma vez ativado, o mecanismo está pronto para receber mensagens dos sistemas de detecção de mau-uso ou anomalia em seus nós, alertando sobre um ataque. Quando um agente de resposta recebe uma mensagem dessas, ele cria um agente de reação direta (`directReactionAgent`). O agente de resposta cria também agentes de alarme (`alarmAgents`), os quais serão responsáveis por comunicar agentes de resposta em outros pontos de resposta sobre a detecção e resposta que está acontecendo.

O mecanismo desenvolvido requer a definição de políticas de resposta para cada pondo de resposta do sistema. Estas polícias devem estabelecer diferentes estratégias de respostas primária e secundária, visando minimizar possíveis riscos de respostas sobre processos válidos. Os pontos de respostas devem ser distinguidos entre as classes: servidor externo, servidor interno, estação de trabalho e componente de rede. Cada uma dessas classes possui a sua própria política de resposta.

2.2.4 Arquitetura Imuno e ADenoIDS

O professor Fabrício de Paula, em sua tese de doutorado [22], reafirma a visão inicial do projeto Imuno de que sistemas de detecção de intrução imunológicos que exploram apenas parcialmente as características do sistema imunológico humano, incorrem em um alto índice de alarmes-falsos (falsos positivos) devido à falta de uma visão geral do sistema biológico.

Considerando uma visão geral do sistema imunológico humano, Fabrício defende a hipótese de que é possível identificar ataques desconhecidos, por meio de análise automática de evidências de intrusão, tornando-os conhecidos. Um ataque desconhecido, portanto, deve permanecer indetectável até o inicio da exploração do alvo, momento no qual o ataque começa a gerar evidências de intrusão suficientes para sua detecção.

Para validar esta hipótese, o trabalho tem o objetivo de construir uma arquitetura de segurança computacional baseada no sistema imunológico humano, através da qual seja possível a identificação precisa de ataques conhecidos, detecção de ataque desconhecidos através de evidências de intrusão, mecanismos de resposta para ambos os tipos de intrusão, além da capacidade de extração de assinaturas de ataques desconhecidos de forma a torná-los conhecidos.

Essa arquitetura foi concebida para proteger um ambiente formado por um conjunto de computadores interligados em rede e com acesso à Internet. A Figura 2.7 ilustra

o ambiente considerado. Neste ambiente, um ataque conhecido pode ser identificado e bloqueado por um IDS baseado em conhecimento, um ataque desconhecido, contudo, pode não ser detectado inicialmente e ter sucesso na invasão da máquina atacada. Apesar de concebida em nível de rede, a arquitetura é diretamente aplicável em nível de *host*, bastando para isso a definição de diferentes assinaturas (baseadas em fluxo de informações, chamadas de sistema, parâmetros de processo, dentre outros).

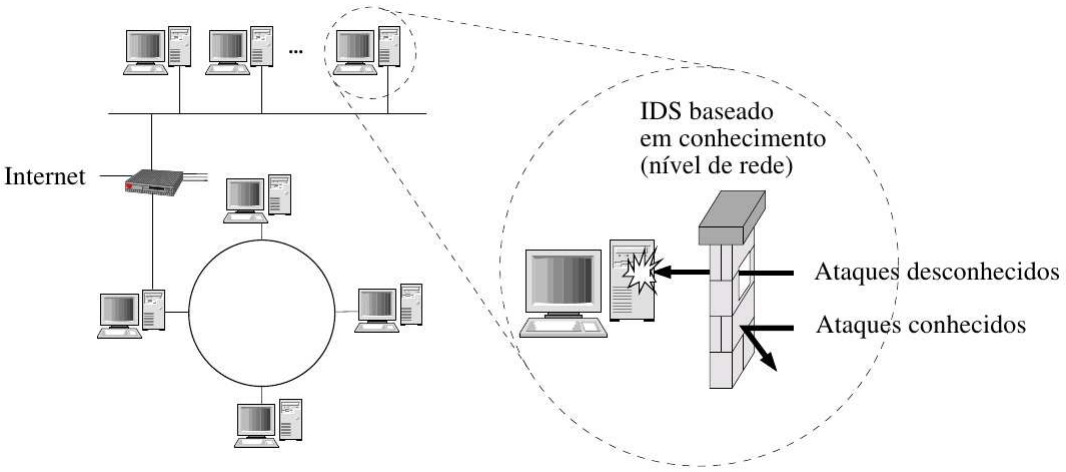


Figura 2.7: Modelo de ameaça considerado em [22].

A arquitetura proposta é uma evolução do modelo inicial do projeto Imuno [23] e o principal aspecto do sistema imunológico considerado foi a sua capacidade de tolerância a invasão. De fato, conforme visto na Seção 2.1, o sistema imunológico humano em muitos momentos é invadido por patógenos que causam danos à saúde antes que haja uma reação que consiga eliminá-los. Após tal invasão bem sucedida, contudo, o sistema imunológico aprende a lidar com este tipo de patógeno, diminuindo ou eliminando o impacto de exposições futuras, e também restaura o organismo protegido a uma condição saudável. Os principais objetivos da arquitetura são:

- Detecção precisa de ataques conhecidos e resposta efetiva contra estes;
- Detecção de ataques desconhecidos através da análise de evidências de intrusão no sistema;
- Habilidade de lidar com ataques desconhecidos de modo a manter o sistema em condições aceitáveis durante a análise das evidências;
- Habilidade de aprendizado a respeito de ataques previamente desconhecidos, de modo a extrair assinaturas destes para detecção futura;

- Habilidade de armazenamento de informações referentes ao ataque de forma a possibilitar análise futura;
- Restauração de partes do sistema afetadas pela intrusão;
- Detecção precisa de ataques que o sistema aprendeu automaticamente a reconhecer.

A Figura 2.8 apresenta a arquitetura proposta. Os retângulos de linha sólida representam os seus componentes e as setas de linha sólida representam o fluxo de informação dentre estes componentes. As setas de linha tracejada representam o fluxo de controle correspondente à ativação de componentes e os retângulos de linha tracejada representam um agrupamento lógico de componentes.

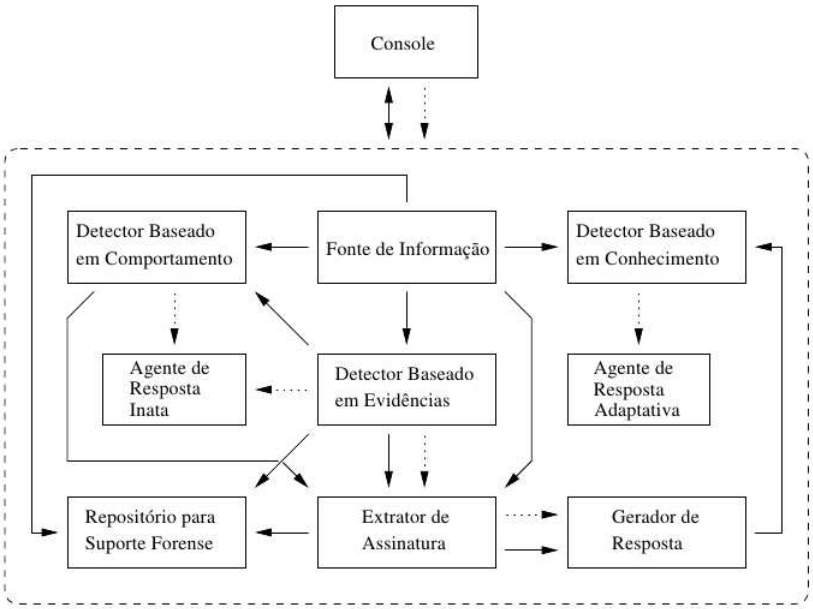


Figura 2.8: Nova Arquitetura de Segurança inspirada nos sistema imunológico humano [22].

Nesta arquitetura, a manipulação de ataques conhecidos é feita através da detecção baseada em conhecimento (análise da ocorrência de assinaturas de ataque conhecidas no fluxo de informação monitorado) associada a uma resposta específica (conjunto de medidas de bloqueio do ataque identificado), realizada pelos componentes Detector Baseado em Conhecimento(1) e Agente de Resposta Adaptativa (2).

Já a detecção de ataques desconhecidos é feita por meio de análise de evidências de intrusão, feita pelo componente Detector Baseado em Evidências (3), através de busca por possíveis violações de políticas de segurança no fluxo de eventos provenientes do componente Fonte de Informação, e pelo componente Detector Baseado em Comportamento

(6), através de diferente análise sobre estas mesmas informações. O Detector Baseado em Comportamento busca nas informações que analisa comportamentos que extrapolem limiares pré-estabelecidos para identificação de uma provável invasão. Este componente recebe constantementes informações do Detector Baseado em Evidências, utilizadas para reforçar suas avaliações.

Para lidar com ataques desconhecidos, o componente Agente de Resposta Inata (5) é acionado na ocorrência de suspeita de invasão, iniciando medidas de contenção capazes de retardar a evolução do provável ataque, sem contudo eliminá-lo, reduzindo desta forma o impacto de atuação sobre falsos-positivos. Ademais, os componentes Extrator de Assinatura (7), Gerador de Respostas (8) e Repositório (9) completam os requisitos necessários para lidar com ataques desconhecidos. O primeiro tem papel de, baseando-se nas informações disponíveis no sistema de segurança, extrair uma assinatura que identifique de maneira precisa o ataque, permitindo fácil identificação em eventuais exposições futuras. O Gerador de assinaturas, por sua vez, deve definir um conjunto de ações de resposta a serem impostas na ocorrência de uma dada assinatura. O último, Repositório para Suporte Forense, é responsável pelo armazenamento seguro das informações coletadas pelo sistema.

Avançando na composição da arquitetura, o componente Console (10) representa a *interface* entre o administrador e sistema de segurança, através da qual é possível o ajuste de parâmetros de configuração e visualização de resultados e alertas do sistema o componente Fonte de Informação. Completando a sua composição, o componente Fonte de Informações (4) tem papel de coleta, armazenamento temporário e disponibilização de dados relevantes ao funcionamento dos componentes na arquitetura proposta.

Apesar de não ilustrado na Figura 2.8, a arquitetura conta ainda com mecanismos de auto-proteção, garantindo que os seus componentes continuem seguros e funcionais mesmo durante a ocorrência de uma invasão.

Esta arquitetura possui clara relação com o sistema imunológico humano. A Tabela 2.2 ilustra algumas analogias entre a arquitetura e o sistema imunológico. Outra visão do relação da arquitetura com o sistema imunológico pode ser feita em termos da abrangência dos seus componentes com o funcionamento dos sistema imunológicos inato e adaptativo. A Figura 2.9 ilustra essa visão.

Para fins de validação das idéias introduzidas através de testes e análises de resultados práticos, esta arquitetura foi parcialmente implementada em um prótotipo de sistema de segurança voltado para identificação e extração de assinaturas de ataques a aplicações do tipo *buffer overflow*. O protótipo desenvolvido foi batizado de ADenoIdS (***A****cquired* ***D****efense System Based on the* ***I****mmune* ***S****ystem*) [5].

[5]Segundo o autor, além do significado direto da sigla, o nome do protótipo também refencia os termos "adenóides", que são nódulos linfáticos que contêm células imunológicas especializadas em proteger o

Sistema Imunológico	**Segurança Computacional**
Console	Imunidade adquirida artificialmente
Fonte de Informação	Fonte de proeínas *self* e *nonself*
Detector Baseado em Comportamento	Detecção de anomalias do sistema inato
Extrator de assinatura	Digestão de antígenos e apresentação de seus fragmentos
Detector Baseado em Conhecimento	Detecção precisa através de células de memória de alta afinidade
Agente de Resposta Adaptativa	Resposta específica do sistema adaptativo
Gerador de respostas	Produção de anticorpos específicos

Tabela 2.2: Analogia entre a arquitetura e o sistema imunológico humano.

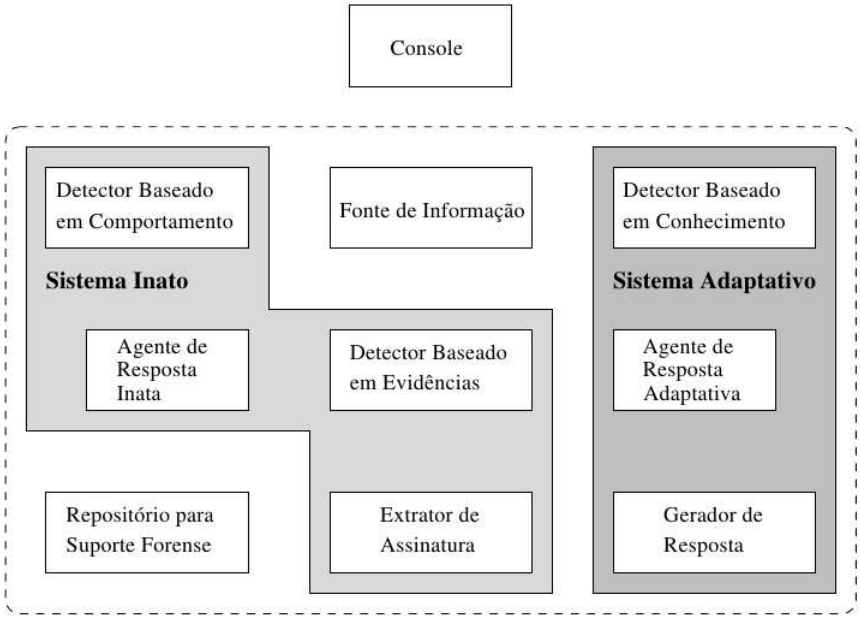


Figura 2.9: Visão dos sistemas inato e adaptativos na arquitetura imunológica.

Módulo ADenoIdS	Relacionamento com a Arquitetura Imuno
ADCON	Console
ADDS	Fonte de Informação
ADEID	Detector Baseado em Evidências
ADBID	Detector Baseado em Comportamento
ADIRA	Agente de Resposta Inata
ADSIG	Extrator de Assinatura
ADFSR	Repositório para Suporte Forense

Tabela 2.3: Relacionamento entre compoentes da Arquitetura Imuno e módulos do protótipo ADenoIdS.

O ADenoIdS foi desenvolvido para monitorar processos de um computador isolado, na busca por evidências de ataques do tipo *buffer overflow* remoto em processos vitimados. Os componentes idealizados na arquitetura da Figura 2.8 foram implementados através de módulos[6]. A Tabela 2.3 relaciona os módulos do protótipo com os componentes da Arquitetura Imuno. Além dos módulos, o protótipo implementou também uma ferramenta batizada de UNDOFS, a qual disponibiliza recuros capazes de restaurar sistemas de arquivo, através de primitivas do tipo *undo* e *redo*.

Diversos testes com diferentes bases de dados foram feitos com o protótipo [22]. O mecanismo de detecção de evidências apresentou taxas de falso-positivos e falsos-negativos iguais a zero. O mecanismo de recuperação foi capaz de restaurar o sistema a um estado funcional, comprovando a sua eficácia, e as assinaturas geradas pelo módulo extrator de assinaturas obtiveram precisa e automática detecção dos ataques em simulações de nova exposição.

Estes bons resultados permitiram ao autor comprovar a factibilidade da detecção de intrusão baseada em evidências, bem como demonstrar a precisão do algoritmo de detecção desenvolvido por ele. Os resultados foram divulgados em publicações científicas [26] [21] e tiveram boa aceitação tanto na área de computação evolutiva quanto na área de segurança de redes.

2.2.5 Framework Imuno

A partir da análise dos trabalhos anteriormente desenvolvidos no projeto Imuno, Martim Carbone notou uma questão comum a todos eles: alta complexidade de implementação dos conceitos propostos. Segundo sua avaliação, o esforço envolvido no processo de desen-

sistema respiratório humano, e IDS (*Intrusion Detection System* - Sistema de detecção de Intrusões).

[6]Três componentes da arquitetura não chegaram a ser implementados no protótipo: Detector Baseado em Conhecimento, Agente de Resposta Adaptativa e Gerador de Respostas.

volvimento dos protótipos de alguma forma diminuia o tempo útil de concepção e avaliação das soluções, representando um considerável ofensor na proposta inicial de desenvolver um sistema imunológico de escopo geral e funcional.

Outra questão observada foi que, apesar de todos os trabalhos terem sido concebidos sobre uma arquitetura comum, explorando diferentes mecanismos de sua composição, as suas implementações utilizavam arquiteturas de implementação de *software* direferentes, fazendo com que não houvesse qualquer tipo de interoperabilidade funcional entre os trabalhos.

Após estudo detalhado das implementações, Carbone constatou ainda que os trabalhos desenvolvidos possuiam complexa manutenção, uma vez que alteravam diversos mecanismos internos do sistema operacional sobre o qual eram implementados.

Diante deste cenário, Carbone propôs então o desenvolvimento de uma base comum de *software* a ser utilizada como plataforma para o desenvolvimento de módulos de segurança imunológicos. Esta ferramenta, portanto, deveria ser um *framework* que implementasse requisitos comuns aos trabalhos de pesquisa em imunologia computacional e disponibilizasse uma API (*Application Programming Interface*) simplificada, que pudesse ser acessada em espaço de usuário pelos demais participantes do projeto.

Devido às caracteristicas dos requisitos dos sistemas imunológicos, tais como capacidade de monitoramento e controle de recursos utilizados por processos, interceptação de chamadas de sistema, captura de tráfego de rede, dentre outros, o *framework*, batizado de *Framework* Imuno, foi implementado todo em nível de *kernel* do sistema operacional GNU/Linux, disponibilizando uma API a ser utilizada pelos processos imunológicos diretamente do espaço de usuário.

O *Framework* Imuno foi projetado para atender os seguintes requisitos:

- Medidas de Prevenção:
  - Controle de acesso
  - Autenticação
  - Filtros de Rede
- Medidas de monitoramento:
  - Monitoramento de chamadas de sistema
  - Estado de processos ativos
  - Captura de tráfego de rede
  - Registro de eventos
  - Monitoramento de consumo de recusos de hardware por processos ativos

- Medidas de Resposta:
  - Controle de Recursos (cotas e garantias)
  - Controle de atributos de processos (prioridade de escalonamento, ID, grupo, etc)
  - Restauração do sistema (restauração de danos provocados por ataques no sistema de arquivos)
- Análise Forense:
  - Sistema de arquivos (acesso ao estado corrente e histórico de alterações)
  - Memória principal (acesso ao estado corrente e *dumps* para disco de regiões específicas para manutenção de histórico de alterações)
  - Tráfego de rede (acesso ao histórico de dados recebidos e enviados via rede)
- Auto-proteção:
  - Proteção de processos imunológicos
  - Proteção de repositórios forenses em disco

Em sua implementação, Carbone utilizou ferramentas disponíveis e consolidadas no Linux, considerando-as parte integrante de sua solução, incorporou ferramentas extenas ao Linux e, para os casos ainda assim não atendidos, desenvolveu ferramentas que atendessem aos requisitos levantados.

Dentre os componentes nativos do Linux utilizados no *Framework* Imuno estão: LSM, Netfilter, `ptrace()`, *syslog*, controle de atributos via chamadas de sistema, `/dev/mem` `/dev/kmem`, *Linux Capabilities System* e SEClvl. As ferramentas externas incorporadas foram: CKRM, RelayFS e UNDOFS. Para completar o atendimento dos requisitos levantados, foram implementados ainda uma infraestrutura de Ganchos Multifuncionais, mecanismos de autoproteção e mecanismos de bloqueio e desbloqueio administrativo. A Figura 2.10 ilustra a arquitetura da solução.

2.2.6 Outros Trabalhos

Diversos outros trabalhos de pesquisa em sistemas computacionais inspirados no sistema imunológico humano estão disponíveis na literatura.

Um dos maiores grupos de pesquisa no segmento de segurança computacional é o grupo liderado por Stephanie Forest, pesquisador com diversos trabalhos de relevância publicados, dentre os quais trabalhos de detecção de anomalias por seleção negativa [27]

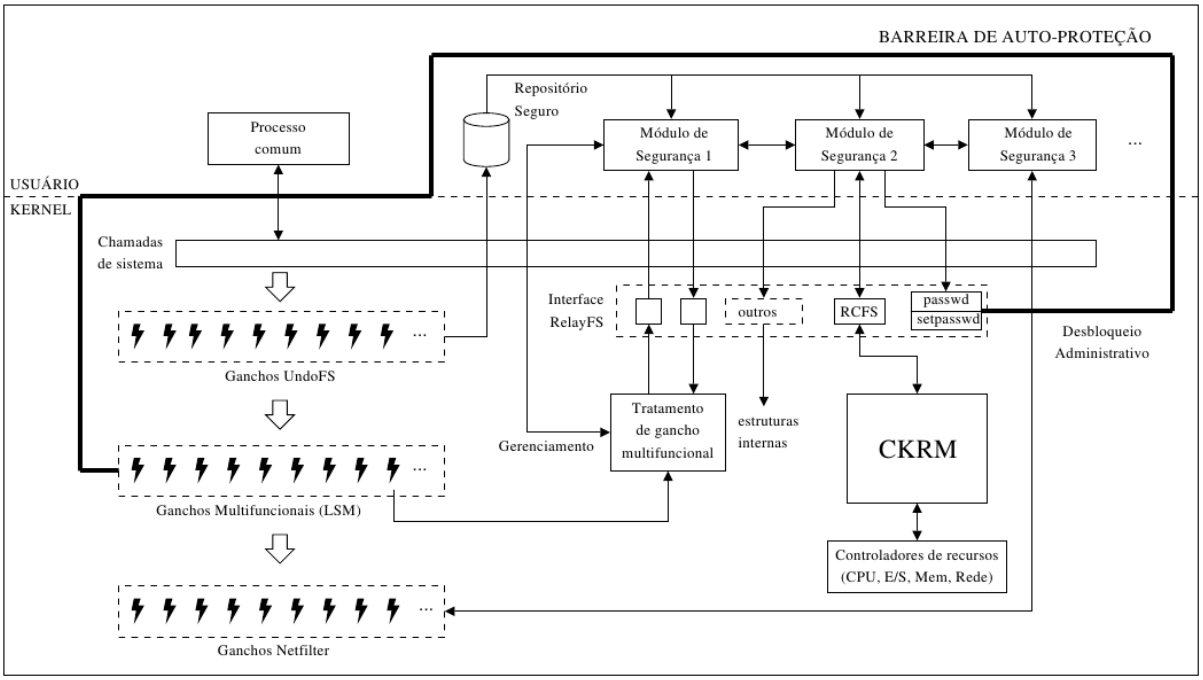


Figura 2.10: Arquitetura detalhada do *Framework* Imuno [29].

[28] [31], propostas de arquitetura de sistemas imunológicos artificiais [39] [34] e trabalhos de mecanismo de resposta automática [60] [59].

O professor Dasgupta, juntamente com sua equipe de pesquisa, possui alguns trabalhos de relevância na área de segurança computacional, nas áreas de detecção de intrusão baseada em anomalias[19] e técnicas de detecção com a utilização de agentes móveis [17] [18].

O trabalho [42], de Kephart et al., propõe ainda um sistema de anti-vírus inspirado em principios de aprendizado do sistema imunológico humano.

Ademais, diversos trabalhos sobre sistemas imunológicos artificiais podem ser encontrados nos anais das edições da conferência ICARIS (*International Conference on Artificial Immune Systems*, especializada em reunir trabalhos de imunologia computacional.

Capítulo 3

Kernel Linux

O conjunto básico de aplicações para o funcionamento de um equipamento de *hardware* é conhecido como Sistema Operacional. O componente fundamental de um sistema operacional é o seu núcleo, *kernel*, em inglês[1]. Os demais programas do sistema operacional são conhecidos como aplicações de usuário e geralmente disponibilizam recursos e funcionalidades para usuários finais do equipamento.

O principal propósito do núcleo de sistema operacional é fazer toda a interação com o *hardware* do equipamento e disponibilizar um ambiente que permita a execução das demais aplicações. Por este motivo, existe uma separação conceitual entre os tipos de código em execução sobre um *hardware*: aplicações em nível usuário, que são aplicações menos privilegiadas que fazem uso do ambiente criado pelo núcleo do sistema operacional; e aplicações em nível de *kernel*, que é o próprio código do núcleo, executando em nível privilegiado e com acesso total aos recursos providos pelo *hardware*[2].

Existe ampla disponibilidade de sistemas operacionais para as mais diferentes arquiteturas de computadores modernos. Dentre os mais conhecidos na indústria, pode-se citar o Windows, OS/X, Solaris, HP/UX e o GNU/Linux [3]. Este último, foi o escolhido para a implementação deste trabalho, por motivos que serão devidamente apontados na Seção 3.1.

Este capítulo também apresentará uma breve revisão de alguns importantes conceitos do *kernel* Linux. A Seção 3.2 introduz aspectos gerais do *kernel*, enquanto as Seções

[1]Neste texto, ambos termos "núcleo" e "*kernel*", são utilizados em referencias ao núcleo de sistemas operacionais.

[2]Os sistemas operacionais dependem de recursos de *hardware* que implementem estes dois níveis de privilégio no acesso a recursos físicos, de tal forma que aplicações em espaço de usuário não possam ultrapassar os limites impostos pelo núcleo.

[3]O termo GNU/Linux é uma referência ao sistema operacional composto pelo *kernel* Linux e aplicações utilitárias do projeto GNU. Neste texto, o termo "Linux" referencia o *kernel* desenvolvido por Linus Torvalds.

3.3, 3.4 e 3.5 discorrem sobre o funcionamento de três importantes funcionalidades do Linux: gerenciamento de processos, chamadas de sistema e gerenciamento de sistemas de arquivos. Estes conceitos são fundamentais no entendimento da solução proposta nesta dissertação.

3.1 Histórico

Motivado pela carência de sistemas operacionais Unix de baixo custo, no ano de 1991 Linus Torvalds iniciou o desenvolvimento do Linux, um sistema operacional baseado em sistemas Unix, sem, contudo, compartilhar código pré-existente.

A principal diferença do sistema operacional Linux com relação aos demais sistemas operacionais existentes àquela época, estava na sua licença de distribuição. O Linux foi inicialmente distribuído com a licença GPL [4]. Esta licença, garante que o código fonte seja distribuído livremente, podendo ser utilizado, modificado e evoluído livremente pelos usuários, desde que respeitada a condição de que os sub-produtos de códigos GPL continuassem sendo distribuídos sob esta mesma licença.

O Linux ganhou rapidamente popularidade no ambiente universitário e, em pouco tempo tinha alcance internacional. Hoje, o Linux é um *kernel* completo, rico em recursos e distribuído para várias plataformas de *hardware* e continua sob constante evolução, através do trabalho colaborativo de milhares de programadores ao redor do mundo, ainda sob forte controle do seu criador e principal mantenedor, Linus Torvalds.

O Linux é, contudo, apenas o núcleo do sistema operacional. O Linux torna-se um sistema operacional completo quando combinado com aplicações em espaço de usuário. Grande parte destas são desenvolvidas pelo projeto GNU (*GNU is not Unix*). Diversas instituições ao redor do mundo reúnem o núcleo Linux com aplicações GNU e outras mais, distribuindo montagens prontas para uso final do GNU/Linux. Estes sistemas são conhecidos como distribuições Linux. Uma lista não exaustiva de distribuições seria: Gentoo, Ubuntu, Mandriva, Suse, Slackware, Debian, dentre outras.

Por tratar-se de um sistema operacional livre, com código fonte disponível e por ser um sistema moderno e amplamente utilizado na academia e na indústria, este trabalho é implementado utilizando este sistema.

3.2 Aspectos Gerais

O GNU/Linux é um sistema operacional multiusuário. Esta característica implica que aplicações de diferentes usuários podem ser executadas de maneira concorrente, sem que quaisquer uma delas tenha conhecimento da existência das demais. Um sistema operaci-

onal multiusuário demanda uma série de cuidados em sua implementação, que garantam um mínimo de segurança para aplicações de diferentes usuários, a saber: proteção de memória, autenticação e controle de acesso.

O GNU/Linux é também um sistema operacional multiprocessado e multiprocessador. Em um sistema multiusuário, espera-se a existência de vários processos de forma simultânea. Portanto, em sistemas com um número de processos maior que o número de CPUs disponíveis, o sistema operacional deve ser capaz de fazer uma divisão deste escasso recurso, preservando a noção de que processos de usuário não têm conhecimento da existência dos demais processos. Esta característica, conhecida como multiprocessamento, onde mais de um processo utiliza "simultaneamente um único processador, também aplica-se em sistemas com mais de um processador, ou multiprocessados, que podem possuir, um número maior de processos que o número de processadores.

É importante observar que o termo "simultaneamente, neste contexto, não está empregado em seu sentido literal, uma vez que um único CPU é capaz de executar uma única instrução em um dado instante. Desta forma, a característica de multiprocessamento de sistemas operacionais é obtida através de uma multiplexação no tempo dos diferentes processos que concorrem pelos recursos de processamento. Esta multiplexação é feita pelo escalonador de processos do sistema operacional. O funcionamento do escalonador do Linux será apresentado na Seção 3.3.1.

Outras importantes características do Linux são: reentrância e preempção. Um *kernel* reentrante permite que mais de uma aplicação de espaço de usuário esteja executando em modo *kernel* em um dado instante. Esta abordagem exige um cuidado especial no sincronismo entre processos, uma vez que pode haver concorrência na utilização de estruturas internas do *kernel*. A preempção, por sua vez, significa que uma rotina em nível de *kernel* pode ser interrompida, de forma a permitir a execução de outras atividades, sendo retomada em momento futuro, criando a ilusão de que múltiplas rotinas de *kernel* são executadas simultaneamente.

Por fim, o *kernel* Linux é do tipo monolítico, contudo, pode admitir a inclusão e exclusão dinâmica de código objeto, através do mecanismo de *Linux Kernel Modules* (LKMs).

3.3 Gerenciamento de Processos

Processos são a abstração fundamental de um sistema operacional. Um processo é uma instância em execução de um determinado programa [16]. Um dado processo possui também uma série de recursos atribuídos a ele, tais como arquivos abertos, sinais pendentes, informações de estado, espaço de memória, uma ou mais *threads* de execução, dentre outros atributos.

O Linux disponibiliza para processos uma abstração de processador e memória virtual, dando a estes a ilusão de que eles são únicos e utilizam o *hardware* exclusivamente. O processador virtual é disponibilizado pelo escalonador de processos, que será detalhado a seguir. A memória virtual permite que processos não tenham acesso ao espaço de endereçamento de outros processos. Diferentes *threads* de um mesmo processo, contudo, compartilham o mesmo espaço de memória.

Todos estes recursos e informações sobre processos manipulados pelo Linux ficam armazenados em estruturas de dados conhecidas como descritores. Um descritor de processo é uma estrutura do tipo `struct task_struct`. Uma listagem parcial dos atributos armazenados nesta estrutura é listado a seguir:

```
struct task_struct {
...
   volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
   void *stack;
   pid_t pid;
   pid_t tgid;
   struct files_struct *files;
   struct signal_struct *signal;
   struct sigpending pending;
   struct sched_entity se;
   int on_cpu;
   int on_rq;
   int prio, static_prio, normal_prio;
   struct task_group *sched_task_group;
   struct css_set __rcu *cgroups;
   struct task_struct __rcu *parent;
   struct list_head children;
   const struct cred __rcu *cred;
   char comm[TASK_COMM_LEN];
...
};
```

Processos podem estar em diferentes estados. Estes estados são descritos abaixo e relacionados na Figura 3.1. O estado de um processo fica armazenado na variável inteira `state`, do seu descritor de processo.

- `RUNNING`: estado em que um processo está pronto para ser executado. O processo pode estar realmente em execução em um processador, ou pode estar aguardando pela sua fatia de tempo de processamento, em uma fila de execução;

- `INTERRUPTIBLE`: estado em que um processo está bloqueado ou "dormindo". Este estado é mantido enquanto um processo aguarda por uma condição necessária à continuação da sua execução, tal como a conclusão da leitura de um arquivo em disco, por exemplo. Quando tal condição torna-se verdadeira, o *kernel* muda o estado do processo para `RUNNING`, transferindo novamente este para uma fila de execução. Processos no estado `INTERRUPTIBLE` podem ser acordados antes da condição pela qual aguardam tornar-se verdadeira, por meio do recebimento de um sinal proveniente de outro processo;

- `UNINTERRUPTIBLE`: estado semelhante ao anterior. Neste estado, contudo, o processo sempre permanecerá bloqueado enquanto aguarda por uma condição, não sendo desbloqueado para responder a sinais vindos de outros processos;

- `ZOMBIE`: estado de um processo logo após a sua finalização, equanto o seu descritor ainda existe. A liberação do descritor de um processo finalizado deve ser feita pelo processo pai, através da chamada de sistema `wait4()`;

- `STOPPED`: estado no qual um processo está paralisado e não pode retomar sua execução. Este estado é obtido quando um processo recebe o sinal `SIGSTOP`.

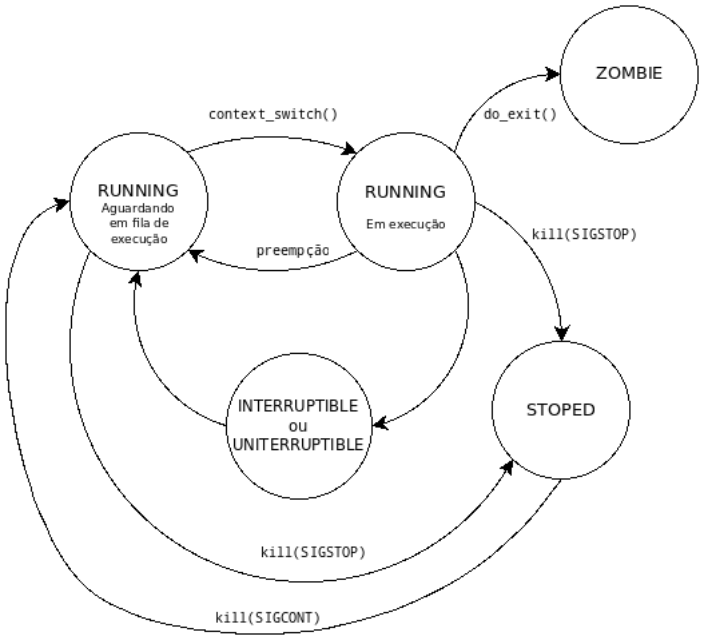


Figura 3.1: Relacionamento entre estados de um processo.

3.3.1 Escalonador CFS

O escalonador de processos do Linux é o componente responsável pela criação da abstração de um processador virtual para um determinado processo. Este componente cria a ilusão de que cada processo possui um processador dedicado.

Na verdade, o trabalho deste componente é interromper um processo em execução e selecionar o próximo processo a ser executado, sendo portanto o coração de um sistema operacional multiprocessado. A escolha do processo é feita através de análise algorítmica de diversos fatores, dentre eles o estado do processo, prioridade de execução e política de escalonamento. A interrupção de um processo, por sua vez, acontece quando a fatia de tempo estabelecida a um processo acaba ou quando um processo sofre uma interrupção.

O algoritmo de escalonamento utilizado no Linux, a partir da versão 2.6.23, é o *Completely Fair Scheduler* (CFS). Este algoritmo tem como objetivo uma divisão justa de tempo entre processos sem, contudo, prejudicar as necessidades de interatividade do sistema. Do ponto de vista de um escalonador, justiça significa uma evolução coerente entre processos concorrentes. O contrário de justiça seria um avanço significativo de um processo em detrimento de outro, que ficaria esperando por um longo período de tempo para utilizar o processador. Sucessor do algoritmo O(1) [6], este escalonador busca também corrigir algumas deficiências de seu predecessor [49].

O CFS possui uma estrutura modular, implementando diferentes classes (ou políticas) de escalonamento. A classe normal de compartilhamento de tempo entre processos é chamada `SCHED_OTHER`. As demais classes disponíveis são: `SCHED_RT` (*real-time*), `SCHED_FIFO` (*First In First Out*) e `SCHED_RR` (*Round Robin*).

O CFS usa uma única estrutura de árvore, do tipo *red-black tree* (árvore RB) [9], por CPU e baseia-se no tempo de execução (`p->se.vruntime`) para rastrear todos os processos executáveis do sistema. Portanto, ideias como: prioridade de filas de execução, identificação de processos interativos e fatias de tempo baseadas em prioridade, todas utilizadas no escalonador O(1), não estão presentes no CFS.

Este escalonador introduz ainda as noções de escalonamento justo em nível de grupos (*group-fair*) ou em nível de usuários (*user-fair*). Justiça de escalonamento em nível de grupo atribui tempo de processamento equivalente a diferentes grupos de processos. Escalonamento justo a nível de usuário, por sua vez, busca um equilíbrio de tempo de execução de processos pertencentes a diferentes usuários do sistema operacional. A Seção 4.5 ilustra a utilização destes conceitos por parte de diferentes subsistemas do *framework* TCG.

Portanto, o CFS foi projetado para contemplar os seguintes objetivos:

- Oferecer bom desempenho interativo ao mesmo tempo em que oferece utilização máxima de CPU;

- Oferecer distribuição justa entre as entidades escalonáveis. Estas entidades podem ser usuários, grupos de usuários, "*containers*" (grupos de processos) e processos;
- Estrutura modular, introduzindo a noção de classes de escalonamento.

A lógica de escalonamento do CFS é a de sempre tentar escalonar o processo cujo atributo `se.vruntime`, que corresponde ao tempo real de execução do processo normalizado pelo número total de processos, possua o menor valor, tentando assim, modelar um sistema de multiprocessamento ideal, onde todos os processos teriam o seu tempo de execução igual a qualquer momento.

A estrutura básica do escalonador é a sua fila de execução, definida através da estrutura `struct rq`. O CFS define uma única fila de execução para cada CPU do sistema, não possuindo a noção de filas diferenciadas de execução.

As filas de execução do CFS são ordenadas de acordo com o valor do atributo `p->se.vruntime`. Este valor, portanto, é utilizado para determinar o posicionamento de novas entidades na árvore RB da fila.

Esta árvore define uma linha de tempo para execuções futuras. O CFS sempre seleciona para escalonamento o elemento mais à esquerda da árvore. Conforme um processo executa, ele é colocado mais a direita da árvore, garantindo, contudo que todos os processos executáveis tornem-se em algum momento o elemento mais à esquerda na estrutura.

Em suma, o CFS permite a execução de um processo por um determinado tempo. Quando um processo escalona (ou um *tick* do escalonador acontece), o uso de CPU daquele processo é contabilizado. O tempo medido é, então, somado ao valor atual do atribuído `p->se.vruntime`. Uma vez que este valor torne-se suficientemente grande para que outro processo se torne o elemento mais à esquerda da árvore RB, o novo elemento é escalonado e o processo atual sofre preempção.

3.4 Chamadas de sistema

Chamadas de sistema representam a *interface* padrão de comunicação entre aplicações no espaço de usuário e núcleo de sistemas operacionais [61]. No Linux, as chamadas de sistema são a única forma válida de comunicação entre processos de usuário e o *kernel*[4].

Chamadas de sistema são invocadas por processos em espaço de usuário através de chamadas de função presentes em bibliotecas de programação, tal como GLIBC [3]. Estas funções intermedeiam o acionamento de chamadas de sistema, tipicamente fazendo alguns

[4]No *kernel* Linux, é relativamente comum a utilização de diferentes sistemas de arquivos virtuais (*pseudo-filesystems*) para troca de informações *kernel-userspace*. Esta abordagem, contudo, utiliza chamadas de sistema em última instância, em operações como abertura, leitura e escrita sobre arquivos do *pseudo-filesystem*.

ajustes sobre os parâmetros informados na chamada de função feita pela aplicação, e acionando as chamadas de sistema. A Figura 3.2 ilustra este processo.

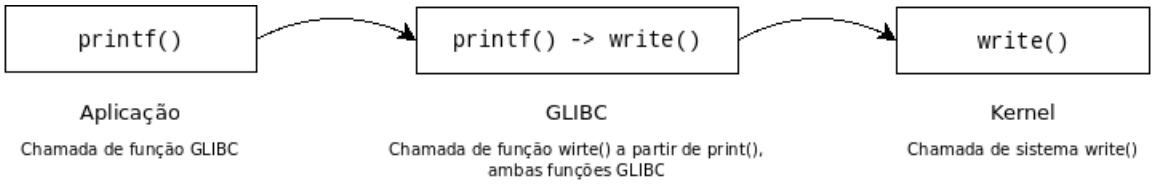


Figura 3.2: Fluxo de execução de uma chamada de sistema.

O Linux dispõe atualmente de 382 chamadas de sistema. A lista completa de chamadas está disponível em [2]. Cada uma destas chamadas possui um escopo específico e bem definido. Um exemplo de implementação de uma chamada de sistema do *kernel* Linux é ilustrado abaixo:

```
asmlinkage long sys_getpid(void)
{
return current->tgid;
}
```

O modificador `asmlinkage` indica ao compilador que a função definida é uma chamada de sistema. O Linux adota ainda o padrão de prefixo "`sys_`" em suas chamadas de sistema. Cada chamada do *kernel* é identificada através de um número. Este número é utilizado por funções em espaço de usuário para identificar a chamada de sistema a ser acionada. Desta forma, é importante que uma chamada sempre possua o mesmo número, independente de compilação, caso contrário aplicações em espaço de usuário podem falhar totalmente em caso de nova compilação do *kernel*.

Como não é possível a aplicações de espaço de usuário fazer chamadas de função de *kernel*, sistemas operacionais lançam mão de recursos de *hardware* para obter o comportamento desejado na *interface* de chamadas de sistema. Em arquiteturas x86[5], o processo é como segue:

- Uma função em espaço de usuário escreve no registrador `eax` do processador o número da chamada de sistema que deseja executar e, em seguida, através de mecanismos de *hardware*, sinaliza ao *kernel* que deseja executar uma chamada de sistema. A passagem de parâmetros segue a mesma abordagem, utilizando outros registradores do processador para armazenamento dos parâmetros (`ebx`, `ecx`, `edx`, `esi` e `edi`);

[5]Em outras arquitetura, o processo é semelhante, guardadas as particularidades de cada processador.

- Uma vez em espaço de *kernel*, a função `system_call()` (tratador de chamadas de sistema) verifica o número da chamada desejada e, caso se trate de uma chamada válida, procede com a execução da chamada desejada;
- Após a execução da função de *kernel* alvo da chamada, o tratador `system_call()` escreve o valor de retorno no registrador `eax`, e sinaliza ao *hardware* para retornar a espaço de usuário;
- Ao ter seu processo novamente escalonado, a função da biblioteca responsável pelo acionamento da chamada de sistema lê o valor retornado e prossegue com a sua execução, alheia à toda a complexidade envolvida no processo de execução da chamada.

A Figura 3.3 ilustra este processo.

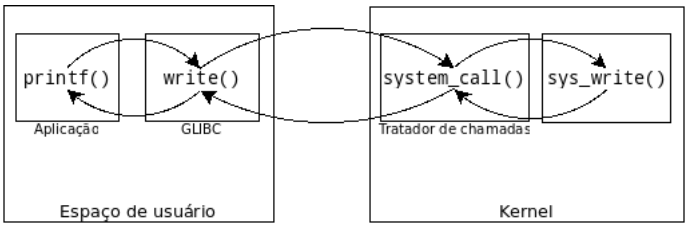


Figura 3.3: Fluxo de execução de uma chamada de sistema Adaptado de [56].

3.5 Sistemas de Arquivos

A implementação de diferentes sistemas de arquivos no *kernel* Linux é feita através de um *middleware* chamado *Virtual File System* (VFS). Este importante componente do Linux disponibiliza uma *interface* comum para a implementação de qualquer sistema de arquivos do sistema operacional, inclusive sistemas de arquivos especiais, como aqueles utilizados por dispositivos ou pelo pelo próprio *kernel*, como o `procfs`, por exemplo.

Esta *interface* implementa uma camada de abstração com todas as operações básicas possíveis em sistemas de arquivos. Para viabilizar tal implementação, todos estes sistemas devem considerar as mesmas abstrações inerentes a sistemas de arquivos introduzidas pelo VFS.

Desta forma, para qualquer chamada de sistema relacionada à manipulação de arquivos, como a chamada `write()`, por exemplo, o trabalho comum à realização da operação, tal como checagem de permissões de acesso, é feito pelo VFS. Os detalhes de implementação inerentes a um tipo específico de sistema de arquivo, por outro lado, é executado pela implementação do sistema em si.

A Figura 3.4 ilustra o fluxo de execução em uma operação de escrita sobre um arquivo no Linux.

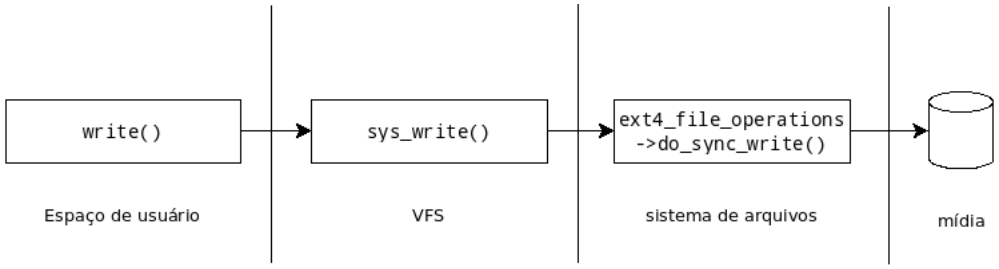


Figura 3.4: Fluxo de execução de uma operação de escrita em disco no sistema de arquivos EXT4.

O comportamento do VFS, e consequentemente de todos os sistemas de arquivos do Linux, considera as seguintes abstrações:

- Sistema de arquivos (***filesystem***): organização hierárquica[6] com uma estrutura específica. Sistemas de arquivos devem conter arquivos, diretórios e outras informações de controle associadas. Objetos deste tipo estão sujeitos a um conjunto de operações, tais como criação, montagem e desmontagem;
- Arquivo (***file***): uma sequência de bytes, onde o primeiro byte marca o início do arquivo e o último byte identifica o seu final. Arquivos possuem atributos, como o seu nome, utilizados na sua identificação e credenciais, utilizados em checagens de controle de acesso. Arquivos também estão sujeitos a algumas operações específicas, tais como abertura, leitura, escrita, dentre outras;
- Diretório (***directory***): arquivos especiais que contêm referências para outros arquivos, utilizados na composição da hierarquia de sistemas de arquivos. Diretórios possuem as mesmas características de arquivos regulares e estão sujeitos às mesmas operações. Ademais, algumas operações sobre arquivos são específicas para criação e remoção de diretórios (`mkdir` e `rmdir`);
- Caminho (***path***): uma sequência de diretórios dentro de um hierarquia, tal como "`/imuno/tcg/suspects`";
- Entrada de diretório (***dentry***): componentes de um caminho. No exemplo anterior, `imuno`, `tcg` e `suspects` são *dentries*;

[6]No Linux, todos os sistemas de arquivos são ancorados, ou "montados", em um determinado ponto de uma hierarquia global única ("`/`"). Desta forma, todos os sistemas de arquivos estão dentro de um único *namespace*.

- ***Inode***: o Linux diferencia as informações associadas a arquivos do conceito de arquivo propriamente dito. Estas informações, conhecidas como *metada*, ficam armazenadas e objetos do tipo *inode*. Estes objetos armazenam informações como credenciais de acesso, data de criação, tamanho, data de modificação, atributos estendidos (`xattr`), dentre outras. *Inodes* também estão sujeitos a um conjunto particular de operações, tais como `link`, `symlink`, `mknod` e `setxattr`;
- Super bloco (***superblock***): estrutura responsável pelo armazenamento de informações de controle (*metadata*) de sistemas de arquivos.

A implementação de sistemas de arquivos no Linux é feita através da definição de *callbacks*[7] para as diversas operações inerentes a cada tipo de objeto do VFS. Um sistema de arquivo deve definir tais funções, e declará-las ao VFS através das seguintes estruturas:

- `struct super_operations`: operações sobre objetos do tipo ***superblock***;
- `struct inode_operations`: operações sobre objetos do tipo ***inode***;
- `struct dentry_operations`: operações sobre objetos do tipo ***dentry***;
- `struct file_operations`: operações sobre objetos do tipo ***file***.

A implementação de sistemas de arquivos através da API do VFS, que manipula os diferentes objetos de sua abstração com o mesmo conjunto de operações, baseado em *callbacks*, permite aos mais diferentes sistemas de arquivos uma implementação com características bastante similares. Atualmente, o Linux possui implementados mais de 50 (cinquenta) tipos diferentes de sistemas de arquivos e, graças à esta abordagem, todos podem interoperar de maneira simples e transparente.

[7] *Callbacks* são trechos de código executáveis que são passados como parâmetro para outros trechos de código. Em C, as *callbacks* são implementadas na forma de ponteiros para funções.

Capítulo 4

Ferramentas de segurança Linux

Em 1996, o Departamento de Defesa dos Estados Unidos publicou um padrão a ser adotado na avaliação do nível de segurança de sistemas computacionais: DoDD 5200.28-STD[51]. Dentre os requisitos de segurança fundamentais de sistemas computacionais apontados pelo documento, estava a necessidade de que tais sistemas deveriam definir uma política de segurança a ser utilizada. Tal política poderia ser de dois tipos: mandatória (MAC) ou discricionária (DAC).

Políticas DAC (*Discretionary Access Control*) consistem em políticas onde a metodologia de controle de acesso é baseada na identidade das entidades do sistema e na sua autorização para acessar os recursos do sistema. Neste sistema, o usuário tem o poder sobre a informação de sua propriedade, sendo capaz de permitir o acesso de terceiros aos objetos de acordo com sua própria vontade.

Já nas políticas MAC (*Mandatory Access Control*), o controle de acesso é feito baseado na comparação de rótulos de segurança atribuidos a objetos do sistema com o nível de acesso atribuído aos sujeitos que tentam acessá-los. Nesta metodologia, um sujeito não pode permitir a outro sujeto o acesso a um determinado objeto, mesmo que o primeiro possua acesso a este. Existe, portanto, a figura de um administrador de políticas de segurança, o qual centraliza a responsabilidade pelo estabelecimento dos níveis de acesso.

Até a versão 2.4 do Linux, o modelo de controle de acesso aos recursos do sistema operacional era meramente discricionário (DAC), baseado em usuários, grupos, listas de controle de acesso e capacidades. O acesso a arquivos e recursos do sistema era feito baseado no conceito de propriedade e atributos de permissão de acesso de terceiros para cada um destes recursos e não existia qualquer tipo de suporte para mecanismos de segurança mais sofisticados.

O Modelo de segurança atual, contudo, é flexível e permite ao administrador do sistema escolher dentre políticas DAC ou MAC, sendo que primeira possui implementação única no *kernel* e é considerada o comportamento padrão do sistema; e a segunda é implementada

na forma de módulos de segurança e é oferecida em diferentes soluções.

A Seção 4.1 deste capítulo apresentará o funcionamento do sistema de controle de acesso do Linux, considerando sua arquitetura e funcionamento padrão (discricionário). A Seção 4.2 apresentará o *framework* LSM (*Linux Security Modules*), utilizado para estender as funcionalidades de controle de acesso do Linux através da implementação de diferentes sistemas de controle de acesso mandatório. Essa Seção também apresentará brevemente as soluções MAC disponíveis no Linux padrão e alguns projetos de pesquisa relacionados.

Este capítulo abordará ainda outros recursos relacionados à segurança no ambiente do sistema operacional Linux, a saber: *SecurityFS*, um sistema de arquivos virtual utilizado como *interface* para módulos LSM (Seção 4.3); *Netfilter*, um *framework* para interceptação e filtro de pacotes de rede (Seção 4.4); e *Task Control Group*, um *framework* para agrupamento de processos e contabilização de uso de recursos por estes grupos, utilizado para monitorar o uso de recursos do sistema operacional e prevenir abusos (Seção 4.5).

4.1 Sistema de Controle de Acesso

O sistema de controle de acesso do Linux considera diferentes objetos em suas checagens de segurança. As checagens são feitas em nível de *kernel* nas situações onde um objeto atua sobre outro.

Estes objetos (***objects***[1]) são diferentes componentes do sistema operacional que podem ser manipulados diretamente por aplicações em espaço de usuário, podendo ser: processos, arquivos e *inodes*, *sockets*, *message queues*, segmentos de memória compartilhada, semáforos e chaves. Estes objetos possuem um conjunto de credenciais (***credentials***), variável de acordo com o tipo de objeto.

As credenciais armazenadas em objetos do tipo arquivo podem variar de acordo com o sistema de arquivos utilizado, podendo incluir as seguintes informações: UID, GID, modo, *Windows user ID*, ACLs, rótulo LSM, bits de escalada de privilégios (SUID e SGID), e bits de capacidades (*capabilities mask*). Já as credenciais de processos podem ser: UID, GID, grupos (`/etc/groups`), chaves e campo LSM. As informações de credenciais de processos ficam armazenadas em sua estrutura `task_struct` através de dois ponteiros para estruturas do tipo `cred`: `real_cred` e `cred`. Estas estruturas possuem a seguinte assinatura[2]:

```
struct cred {
 ...
```

[1] Os termos em fonte <***negrito itálico***> representam a nomenclatura utilizada no Linux para identificar os componentes do seu sistema de controle de acesso.

[2] Apenas campos que armazenam informações de credenciais estão exibidos abaixo. Informações de controle estão omitidas.

```
 kuid_t          uid;              /* real UID */
 kgid_t          gid;              /* real GID */
 kuid_t          suid;             /* saved UID */
 kgid_t          sgid;             /* saved GID */
 kuid_t          euid;             /* effective UID*/
 kgid_t          egid;             /* effective GID*/
 kuid_t          fsuid;            /* UID for VFS ops */
 kgid_t          fsgid;            /* GID for VFS ops */
 unsigned        securebits;       /* SUID-less sec management */
 void            *security;        /* subjective LSM security */
 struct group_info *group_info;    /* groups for euid/fsgid */
 struct user_struct *user;         /* real user ID subscription */
 kernel_cap_t cap_inheritable;     /* caps children can inherit */
 kernel_cap_t cap_permitted;       /* caps we're permitted */
 kernel_cap_t cap_effective;       /* caps we can actually use */
 kernel_cap_t cap_bset;            /* capability bounding set */
 unsigned charjit_keyring;         /* keyring */
 struct key *thread_keyring;       /* this thread's keyring */
 struct key *request_key_auth;     /* assumed request_key authority */
 struct thread_group_cred *tgcred; /* thread-group shared credentials */
 ...
};
```

Um subconjunto das credenciais de um objeto, o seu ***objective context***, é utilizado nas avaliações de controle de acesso do Linux quando tal objeto é acessado. Em sistemas de arquivo Linux o contexto objetivo é representado pelo conjunto UID e GID (*Group IDentification* - IDentificação de Grupo). Em processos, o contexto objetivo é representado pela estrutura apontada pelo ponteiro `struct cred *real_cred`.

Outra definição importante considerada é a de sujeito (***subject***), que representa uma entidade que está exercendo uma ação sobre outro. O exemplo mais direto de sujeito são os processos do sistema operacional, uma vez que tratam-se de instâncias em execução que podem atuar sobre outros objetos. Contudo, outros objetos dos sistema operacional também podem, em algumas situações, ser considerados sujeitos [40]. As ações (***actions***) de sujeitos sobre objetos podem ser operações de leitura, escrita, criação ou remoção de arquivos, criação de processos, sinalização, dentre outras.

Quando sujeitos atuam sobre objetos, um outro subconjunto de suas credenciais, ***subjective context***, é utilizado nas avaliações de controle de acesso. O *subjective context* de um processo é armazenado na estrutura apontada pelo ponteiro `struct cred *cred`.

A avaliação de controle de acesso do *kernel* é feita através de um cálculo envolvendo os

componentes anteriormente apresentados (*subject*, *object*, *action*, *subjetive context*, *objetive context*) e um conjunto de uma ou mais regras (***rules***) aplicáveis ao contexto. Estas regras podem ser dos tipos DAC, que ficam armazenadas em cada objeto, ou MAC, onde existe um conjunto de regras definido para o sistema operacional como um todo.

O fluxo de avaliações do sistema de controle de acesso do Linux é ilustrado na Figura 4.1.

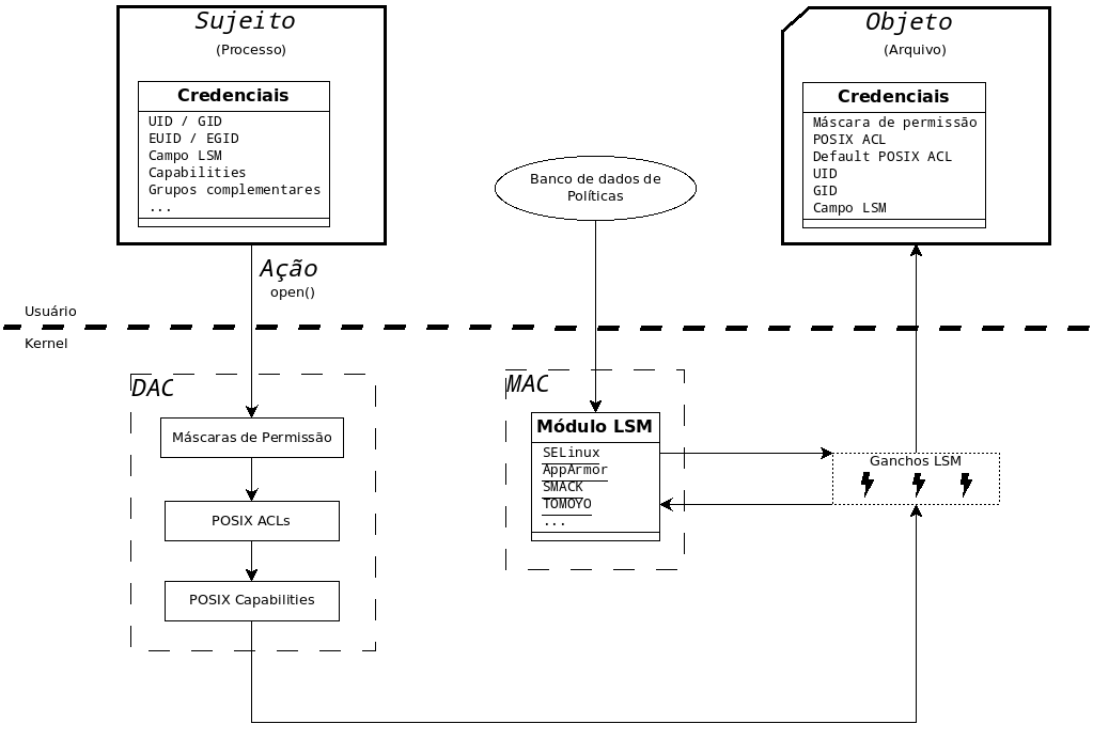


Figura 4.1: Funcionamento do sistema de controle de acesso do Linux.

As regras do DAC do Linux são divididas em três diferentes soluções: máscaras de permissão, ACLs (*Access Control Lists*) e *POSIX Capabilities*. O funcionamento destas implementações DAC será brevemente apresentado a seguir.

Já as regras MAC são implementadas de maneira modular através do *framework* LSM, o qual será apresentado em detalhes na Seção 4.2.

É importante notar que as regras MAC são avaliadas somente após as regras DAC. Ou seja, para os casos em que uma das regras DAC impeça o acesso a um objeto por um sujeito, o acesso será negado mesmo que as regras MAC não ofereçam restrição ao acesso. Portanto, os sistemas MAC disponíveis para Linux são do tipo restritivo: são capazes de negar acesso a objetos em pontos onde tais acessos estavam prestes a ser autorizados pelo controle de acesso DAC. O contrário, contudo, autorizar acessos negados nas avaliações

DAC, não é possível a priori.

4.1.1 Discretionary Access Control - DAC

Máscaras de permissão

Máscaras de permissão são o tipo mais básico de DAC no Linux. Estas máscaras podem ser vistas como uma lista de controle de acesso (ACL) simplificada.

Nesta implementação, os objetos carregam em suas credenciais uma máscara simples, de 16 bits, que regula as permissões de acesso a estes objetos por diferentes classes de sujeitos. Como na arquitetura do Linux todos os objetos são definidos como arquivos, todos eles possuem uma estrutura do tipo `inode`. Esta máscara de 16 bits, portanto, fica armazenada na estrutura `inode` do objeto, em uma variável de nome `mode`. Esta variável é do tipo `short int`[3].

Nestas máscaras, os bits determinam o perfil de acesso autorizado (leitura, escrita e execução) para três classes distintas de sujeitos (*user*[4], *group*[5] e *other*[6]).

Os três bits menos significativos representam permissões de execução, escrita e leitura (partindo do bit menos significativo para o mais significativo) para sujeitos do tipo *other*. Os dois grupos de três bits subsequentes representam as mesmas permissões para sujeitos do tipo *group* e sujeitos do tipo *user*, respectivamente. Cada bit com valor igual a um na máscara significa que a permissão deve ser concedida.

Os três bits subsequentes aos nove descritos anteriormente, representam, também do menos significativo para o mais significativo, as opções: *sticky bit*, SGID e SUID. A função do *sticky bit* varia de acordo com o tipo de objeto e sistema de arquivos, sua aplicação mais usual é garantir que apenas proprietários de arquivos possam apagá-los em ambientes onde todos podem ler e escrever, como por exemplo no diretório `/tmp`. Os bits SUID e SGID, por sua vez, permitem que, ao serem executados, estes objetos gerem credenciais com UID e GID iguais às da credencial do objeto, e não do sujeito que o executou, ou seja, executa o arquivo com as permissões do proprietário, e não do processo que o invocou. Esta funcionalidade e suas aplicações serão detalhadas na Seção POSIX Capabilities.

Os quatro últimos bits utilizados na variável `mode` identificam o tipo do objeto, que pode ser: *regular file*, *directory*, *link*, *socket*, *block device*, *character device* ou FIFO.

Todas essas máscaras estão definidas no arquivo `/linux-src/include/uapi/linux/stat.h`. Suas representações em base numérica octal[7] são as seguintes:

[3]Tamanho de palavra de 16 bits em CPUs de 32 bits.
[4]Sujeito com UID igual ao do objeto.
[5]Sujeito com GID igual ao do objeto
[6]Sujeito com UID e GID diferentes dos armazenados nas credenciais do objeto
[7]A representação em base numérica octal é comumente utilizada por facilitar a rápida interpretação

- 0140000: objeto é do tipo *socket*;
- 0120000: objeto é do tipo *Link*;
- 0100000: objeto é do tipo *regular file*;
- 0060000: objeto é do tipo *block device*;
- 0040000: objeto é do tipo diretório;
- 0020000: objeto é do tipo *character device*;
- 0010000: objeto é do tipo FIFO;
- 0004000: SUID habilitado;
- 0002000: SGID habilitado;
- 0001000: *Sticky bit* habilitado;
- 00700: operações de leitura, escrita e execução permitidas para *user*;
- 00400: operações de leitura são permitidas para *user*;
- 00200: operações de escrita são permitidas para *user*;
- 00100: operações de execução são permitidas para *user*;
- 00070: operações de leitura, escrita e execução são permitidas para *group*;
- 00040: operações de leitura são permitidas para *group*;
- 00020: operações de escrita são permitidas para *group*;
- 00010: operações de execução são permitidas para *group*;
- 00007: operações de leitura, escrita e execução são permitidas para *other*;
- 00004: operações de leitura são permitidas para *other*;
- 00002: operações de escrita são permitidas para *other*;
- 00001: operações de execução são permitidas para *other*;

da máscara, uma vez que cada dígito representa um bloco de três dígitos de sua composição. Nesta representação, o primeiro dígito zero apenas indica que o número está escrito em base octal

POSIX Access Control Lists - ACLs

A implementação de POSIX ACLs no Linux tem como objetivo definir um melhor controle DAC através de um esquema mais elaborado de listas de controle de acesso.

Nesta solução, dois tipos diferentes de ACL são permitidas: ACL relacionada diretamente ao objeto e ACL relacionada à criação de novos objetos dentro de um diretório (aplicável somente em objetos do tipo diretório). Para cada tipo de ACL, pode ser definido um conjunto de regras. Cada regra é representada por uma entrada (*entry*) daquela ACL.

Uma entrada estabelece permissões de acesso (leitura, escrita e execução) para um tipo de sujeito. Cada entrada possui um rótulo (*tag*) que indica o contexto de aplicação daquela regra. Uma entrada pode possuir ainda um qualificador (*tag qualifier*) que representa um usuário (UID) ou grupo (GID) ao qual aquela regra se aplica.

A implementação Linux do POSIX ACLs introduz duas novas variáveis nas credenciais de objetos do tipo `inode`, uma para cada tipo de ACL do padrão, são elas: `struct posix_acl *i_acl` e `struct posix_acl *i_default_acl`.

A estrutura `posix_acl` possui uma lista de variáveis do tipo `struct *posix_acl_entry`. Esta última estrutura representa uma entrada do POSIX ACL, armazenando os seus atributos *tag*, *tag qualifier* e permissões de acesso. A estrutura possui a seguinte declaração:

```
struct posix_acl_entry {
   short           e_tag;
   unsigned short  e_perm;
   union {
      kuid_t       e_uid;
      kgid_t       e_gid;
   };
};
```

Os seguintes valores são válidos para o atributo `e_tag`:

- ACL_USER_OBJ: permissões definidas na entrada aplicam-se a sujeitos do proprietário do objeto
- ACL_USER: permissões definidas na entrada aplicam-se a sujeitos com UID igual ao definido no atributo (*tag modifier*) `e_uid`
- ACL_GROUP_OBJ: permissões definidas na entrada aplicam-se a sujeitos que fazem parte do mesmo grupo do objeto

- ACL_GROUP: permissões definidas na entrada aplicam-se a sujeitos que tenham GID igual ao definido no *tag modifier* `e_gid`
- ACL_MASK: permissões definidas na entrada representam o máximo acesso permitido a objetos do tipo ACL_USER, ACL_GROUP_OBJ ou ACL_GROUP
- ACL_OTHER: permissões definidas na entrada aplicam-se aos objetos que não se enquadram no perfil de nenhuma outra entrada

Uma ACL válida deve conter exatamente uma entrada do tipo ACL_USER_OBJ e uma entrada do tipo ACL_GROUP_OBJ. Entradas do tipo ACL_USER e ACL_GROUP são opcionais e podem aparecer em qualquer número em uma ACL. Sempre que pelo menos uma entrada desses dois últimos tipos compor uma ACL, uma entrado do tipo ACL_MASK passa a ser obrigatória, em casos contrário este tipo de entrada é opcional.

As seguintes permissões de acesso são válidas para o atributo *e_perm*:

- ACL_READ: leitura
- ACL_WRITE: escrita
- ACL_EXECUTE: execução

POSIX Capabilities

O modelo de controle de acesso discricionário tradicional do Linux, composto inicialmente por máscaras de permissão e, posteriormente refinados através do uso de ACLs, avalia a legitimidade de um acesso baseado na ideia de propriedade, através da noção da existência das entidades usuários e informação, onde um (processo) usuário é proprietário da sua informação e tem poderes ilimitados sobre ela, podendo inclusive permitir ou negar acesso à ela para terceiros.

Neste modelo existe ainda uma outra categoria de processos: a dos processos privilegiados (aqueles onde o UID do usuário é igual a 0, ou seja, aqueles pertencentes ao super-usuário, conhecido como usuário *root*). Um processo privilegiado consegue ignorar todas as checagens de permissão do *kernel* e ter acesso a virtualmente todas as informações do sistema.

Em alguns momentos, contudo, processos de usuário precisam fazer acesso a informações privilegiadas do sistema, sem contudo terem ou, por questões de segurança, poderem ter permissão de acesso à elas. Por exemplo, um usuário deve ter a habilidade de alterar a sua própria senha de acesso. No Linux, o banco de dados de senhas consultado por aplicações de autenticação fica localizado no arquivo `/etc/shaddow`. Este arquivo contém a senha de todos os usuários do sistema, inclusive a senha do superusuário e, por

razões óbvias de segurança, um usuário comum não possui acesso de leitura ou escrita sobre este arquivo.

Para solucionar este impasse, o Linux mantém em suas credenciais um atributo (bit *setuid*) que permite a arquivos executáveis de usuário serem executados no contexto de outro usuário (UID efetivo - EUID). Desta forma, uma aplicação do usuário pode intermediar o acesso a objetos de sistema aos quais seu proprietário não possua acesso, mas, em determinado momento, precisa utilizá-los legitimamente, nos casos em que o seu EUID for igual ao do superusuário, ou seja, zero.

Esta solução, contudo, apesar de ser amplamente utilizada, possui uma fragilidade: mesmo quando processos precisam de apenas alguns privilégios de *root*, eles sempre recebem todos os seus privilégios, o que certamente é um exagero. Uma consequência negativa desta característica é o fato de que falhas em aplicações SUID *root* podem ser exploradas por usuários maliciosos para obtenção de acesso total, comprometendo a segurança do sistema como um todo.

Diante deste cenário, a partir da versão 2.2, o Linux passou a dividir os privilégios atribuídos ao usuário *root* em subgrupos, conhecidos como *capabilities* [8]. Estes grupos são atribuídos a processos e podem ser habilitados e desabilitados independentemente.

A implementação de POSIX Capabilities permite, portanto, que processos com UID (ou SUID) zero sejam executados com capacidades reduzidas, limitando o poder de acesso destes aos recursos do sistema operacional, garantindo uma granularidade ainda mais fina no controle de acesso discricionário padrão do Linux.

Esta implementação é feita a partir da adição das chamadas de sistema `capset()` e `capget()`, adição de informações nas credenciais de processos e arquivos e, diversos pontos de checagem espalhados pelo *kernel*.

As informações de credenciais adicionadas aos processos estão na forma de variáveis do tipo `kernel_cap_t`[8] adicionadas à estrutura de credenciais (`struct cred`). Processos carregam três destas variáveis, representando os seguintes conjuntos de *capabilities*: *effective*, *permitted* e *inheritable*: O primeiro conjunto é aquele que é efetivamente avaliado pelas checagens de permissões do *kernel*. O segundo representa o conjunto de *capabilites* que um processo pode habilitar, ou seja, tornar efetivo, via chamada de sistema `capset()`. Já o conjunto *inheritable capabilities* representa o conjunto de *capabilites* que podem ser repassadas a um processo filho.

O VFS (*Virtual File System*) permite que *inodes* possuam em suas credenciais uma máscara identificando o seus conjuntos de *capabilities*, de tal forma que, ao serem executados via *syscall* `execve()`, o novo processo criado possua suas capacidades como uma composição entre capacidades do processo que o criou (*inheritable capabilities*) e as capaci-

[8]Um conjunto de *capabilities* é representado por um inteiro de 32 bits. Na versão 3.7, o Linux dispõe de 36 *capabilities*, definidos no arquivo `linux-src/include/capabilities.h`.

dades armazenadas no seu *inode*. A metodologia de cálculo desta composição é detalhada em [35].

4.1.2 Mandatory Access Control - MAC

O Linux admite diferentes soluções de controle de acesso mandatório (MAC). Estas soluções são implementadas através da utilização do *framework* LSM (*Linux Security Modules*). O LSM e os módulos de segurança existentes atualmente no *kernel* Linux serão apresentados na Seção 4.2.

Existem ainda outras soluções do tipo MAC em desenvolvimento em projetos de pesquisa e que não fazem parte da versão oficial do Linux. Uma breve listagem será apresentada da Seção 4.2.6

4.2 LSM - Linux Security Modules

Até as versões anteriores ao Linux 2.4, apesar de diversas soluções MAC serem pesquisadas e propostas para compor o *kernel* [44] [53] [55], nenhuma solução havia sido aceita e incorporada ao código oficial do Linux, que permanecia limitado às soluções DAC apresentadas na Seção anterior. A principal razão para esta realidade estava na falta de consenso entre desenvolvedores e pesquisadores de qual seria a melhor ou mais geral solução de segurança.

Diante desta falta de consenso, o mantenedor do Linux, Linus Torvalds, defendeu a ideia de que deveria existir um mecanismo que permitisse uma implementação modular de diferentes modelos de segurança. Neste contexto, foi incorporado ao Linux um *framework* de segurança capaz de disponibilizar uma API genérica para o desenvolvimento destes diferentes modelos: o LSM (*Linux Security Modules*) [64].

A abordagem adotada na concepção do LSM[9], foi a de intermediar o acesso aos objetos do *kernel*, permitindo que módulos de segurança (módulos LSM) avaliassem se aqueles objetos poderiam ou não ser acessados, segundo o critério de cada solução. Para isso, o LSM implementou ganchos (*hooks*) em pontos imediatamente anteriores aos pontos de acesso aos objetos. Estes ganchos desviam o fluxo de execução para funções de avaliação implementadas pelos módulos de segurança, que podem permitir ou negar o acesso aquele objeto. A Figura 4.2 ilustra este processo.

Além desta intermediação no acesso a objetos de segurança, o LSM também incluiu campos de segurança opacos nas credenciais de diversos objetos do *kernel*, de tal forma que os módulos LSM possam atualizá-los em rotinas de auditoria de segurança.

[9]Neste texto, o termo LSM é adotado para referenciar o *framework* LSM.

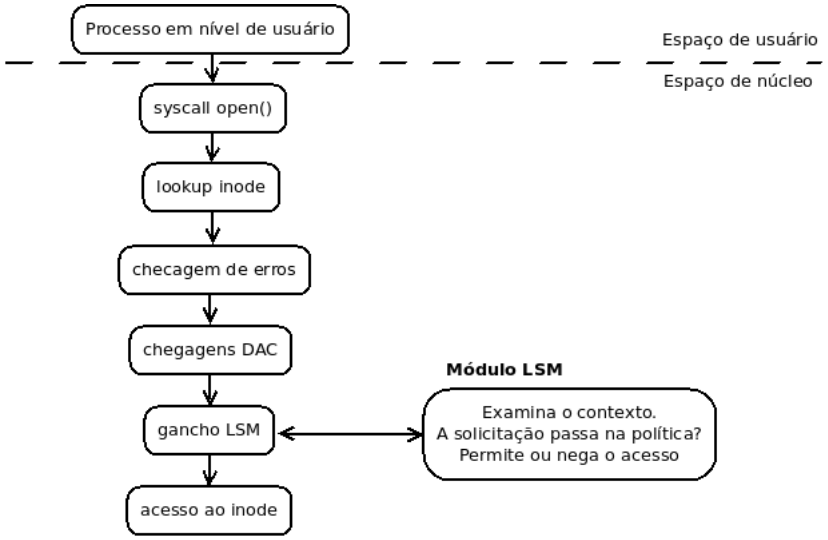


Figura 4.2: Arquitetura de Ganchos LSM.

Os ganchos LSM são, em sua maioria, do tipo restritivos, ou seja, estão distribuídos de tal forma em pontos do *kernel* que o módulo LSM podem negar o acesso a objetos onde este acesso estava prestes a ser autorizado. Em situações onde o *kernel* nega o acesso a um determinado objeto, o *framework* LSM pouco oferece mecanismos para que os módulos sejam consultados e, eventualmente, permitam o acesso anteriormente negado (ganchos permissivos).

Os ganchos do LSM nada mais são que chamadas de funções do LSM espalhadas por todo o *kernel*, em pontos com as características descritas acima, que possuem *callbacks* para funções tratadoras implementadas pelos módulos de segurança. Por exemplo, quando uma aplicação no espaço de usuário deseja criar um novo processo, ela invoca a chamada de biblioteca `fork()`, que por sua vez utiliza a chamada de sistema `sys_fork()`. Esta chamada de sistema utiliza a função `do_fork()` que, em última estância, chama a função `copy_process()` para criar o novo processo. Nesta última função, imediatamente antes da criação do novo processo, é feita uma chamada para a função `security_task_create()`, que é um gancho LSM. Abaixo a sua implementação:

```
/*
 * linux-src/kernel/fork.c
 */
static struct task_struct *copy_process(unsigned long clone_flags,
          unsigned long stack_start, struct pt_regs *regs,
          unsigned long stack_size, int __user *child_tidptr,
          struct pid *pid, int trace)
```

```
{
int retval;
struct task_struct *p;
...
retval = security_task_create(clone_flags); /* Gancho LSM */
if (retval)
return ERR_PTR(retval);
...
/* Copia o Processo */
...
return p;
}
```

A função `security_task_create()`, simplesmente chama a *callback* tratadora deste gancho e retorna o valor por ela retornado:

```
/*
 * linux-src/security/security.c
 */
int security_task_create(unsigned long clone_flags)
{
        return security_ops->task_create(clone_flags);
}
```

O valor de retorno da função tratadora do gancho, implementada pelo módulo LSM, é testado no ponto do *kernel* onde o gancho foi implantado e desviou o fluxo de execução. Caso este valor seja diferente de zero, a operação que estava preste a ser executada não acontece (acesso negado). Ou seja, o módulo LSM é capaz de negar uma operação que já havia sido autorizada pela avalização do *kernel* e estava prestes a ser executada.

Todas as *callbacks* dos módulos LSM são registradas na variável global `security_ops`, do tipo `struct security_operations`, na forma de ponteiros para funções. O LSM permite apenas a definição de uma função tratadora (*callback*) para cada gancho do *kernel*, e esta função deverá ter exatamente a mesma assinatura do gancho. Estas funções tratadoras são definidas no momento do registro do módulo LSM junto ao *kernel*, o qual acontece no momento da inicialização do sistema operacional.

Um módulo de segurança deve, portanto, definir todos os ganchos que irá tratar através de uma variável tipo `struct security_operations` e registrá-la junto ao *kernel*, através da função `register_security()`. Esta função faz com que a variável global `security_ops` referencie a variável `struct security_operations` do módulo de segurança, conforme abaixo:

```
/**
 * linux-src/security/security.c
 * @ops: ponteiro para struct security_options do módulo a ser registrado
 */
int __init register_security(struct security_operations *ops)
{
...
        security_ops = ops;
        return 0;
}
```

Uma possível definição de `struct security_operations` de um módulo de segurança de nome `modlsm` e com *callback* definida apenas para o gancho `inode_mkdir()`, portanto, seria como:

```
struct security_operations modlsm_security_ops =
{
        .name                   = "modlsm",
        .inode_mkdir            = modlsm_mkdir,
};
```

O LSM não permite a execução concomitante de mais de um módulo de segurança, tampouco permite o carregamento de módulos de segurança em tempo de execução, através de módulos dinamicamente carregáveis (*Linux Kernel Module* - LKM)[10].

Na versão 3.7.6 do Linux, o LSM possui 188 ganchos, divididos em categorias conforme abaixo:

- Ganchos para operações de execução de programas
- Ganchos para operações com sistema de arquivos
- Ganchos para operações com *inodes*
- Ganchos para operações sobre arquivos
- Ganchos para operações com processos
- Ganchos para *Netlink messaging*

[10]O projeto original do LSM permitia que um módulo LSM fosse criado a partir de um LKM (*Linux Kernel Module*) e registrado no LSM em tempo de execução. Contudo, a partir do kernel 2.6.24, este símbolo deixou de ser exportado e a interface com o LSM passou a ser estática. Desta forma, todos os módulos LSM passaram a ter a necessidade de serem compilados internamente (*built in*) no *kernel*.

- Ganchos para *Unix domain networking*
- Ganchos para operações com *sockets*
- Ganchos para operações XFRM
- Ganchos para mecanismos de comunicação interprocessos (IPC).
- Ganchos para auditoria de segurança

Os módulos LSM devem implementar suas *interfaces* com o espaço de usuário através de sistema de arquivos virtual, do tipo *securityfs*, de tal forma que parâmetros de funcionamento possam ser ajustados e dados possam ser lidos. A API do *Security FS* para a criação destas *interfaces* ser apresentada na Seção 4.3.

Nas próximas Seções, serão apresentadas alguns módulos de segurança implementados utilizando o LSM e suas principais funcionalidades serão analisadas.

4.2.1 SELinux

O SELinux (*Security Enhanced Linux*) é um LSM que implementa diferentes modelos de MAC no Linux. Este LSM foi desenvolvido inicialmente por pesquisadores do NSA (*National Information Assurance Research Laboratory* [10]), como implementação de uma arquitetura flexível de controle de acesso mandatório desenvolvido por aquele mesmo grupo, chamada Arquitetura Flask [44].

A arquitetura Flask partiu da necessidade de um modelo de segurança flexível, capaz de suportar diferentes políticas de segurança. Seu principal objetivo era disponibilizar um sistema MAC capaz de atender diferentes sistemas computacionais com diferentes requisitos de segurança, partindo da hipótese de que nenhuma política MAC isolada seria capaz de atender a todos os requisitos de segurança de todos os sistemas possíveis [46]. Para atender a este objetivo, a arquitetura baseia-se em um modelo com clara separação entre o mecanismo de controle de acesso (*enforcement mechanism*) e lógica de políticas de segurança. Desta forma, a arquitetura oferece uma *interface* onde é possível a definição de diferentes políticas especificadas por administradores de segurança.

O protótipo inicial do SELinux foi desenvolvido diretamente no *kernel* Linux e sua distribuição era feita através de *patches*. Posteriormente, o SELinux motivou a criação e inclusão do *framework* LSM no *kernel* Linux [64]. O SELinux foi então reimplementado utilizando o *framework* LSM e tornou-se o primeiro módulo LSM a fazer parte das versões oficiais do Linux em dezembro de 2004, no Linux 2.6.

Respeitando a arquitetura Flask, o SELinux é dividido em dois componentes: *Security Server* (SS) e o *Enforcement Mechanism*. Este último atua no controle de acesso de

diferentes subsistemas do Linux, tais como operações sobre processos, arquivos e sockets, através da utilização dos ganchos disponibilizados pelo LSM.

O SS, por sua vez, é responsável pela lógica de decisão de controle de acesso, através da implementação das políticas de segurança definidas pelo administrador do sistema. O SELinux permite a utilização de diferentes SS, onde cada um implementa um modelo de MAC. Os modelos de MAC aplicados pelo SELinux, portanto, podem ser facilmente alterados através da definição de novas políticas de segurança.

As decisões de segurança tomadas pelo SS são baseadas em contextos de segurança (*security contexts*), os quais são representados por rótulos (*labels*). Um contexto de segurança é um tipo de dado, independente de política, que pode ser manipulado por diversas partes do sistema e que é interpretado pelo SS em suas tomadas de decisão. Um contexto deve conter todos os atributos de segurança relacionados ao objeto rotulado que são necessários ao processo de tomada de decisão de uma política.

Outro tipo de dado independente de política de segurança definido pelo SELinux é o *security identifier* (SID). SIDs são definidos em tempo de execução e relacionam objetos a *security contexts*. SIDs associados a objetos são utilizados como base para tomada de decisão pelo SS.

O SELinux é distribuído com um SS padrão, o qual implementa os modelos de segurança TE (*Type Enforcement*) e RBAC (*Role-Base Access Control*) [58]. Apesar de estas definições não serem estritamente necessárias, uma vez que um administrador pode, a partir de definição de políticas, definir totalmente um SS que implemente o modelo MAC que melhor lhe convir, este SS padrão é incluído por representar um modelo suficientemente geral para suportar diversos requisitos de segurança.

O RBAC do SELinux mantém contextos de segurança baseados em três atributos de segurança: identidade (*identity*), papel (*role*) e tipo (*type*). Todos os processos do sistema operacional são associados a identidades (não necessariamente o UID do processo que os criou). Um papel é utilizado para definir ações permitidas aos usuários. O atributo tipo é utilizado para expressar controles de acesso mais refinados.

As políticas de segurança a serem aplicadas em *kernel* pelo *enforcement mechanism* são definidas em arquivos de texto. Uma vez definidas, políticas podem ser ajustadas para atender a requisitos específicos de segurança. O RBAC e TE do SELinux são distribuídos juntamente com o módulo LSM com este propósito.

Os arquivos de configuração de políticas são escritos utilizando uma linguagem própria do SELinux, e devem ser compilados para serem carregados pelo SS.

O TE define tipos de objetos e classifica-os em classes. Operações permitidas entre objetos de uma classe são definidas da seguinte forma:

```
allow type_1 type_2:class { perm_1 ... perm_n }
```

O TE define ainda algumas transições automáticas entre tipos em algumas situações. Define também rótulos padrão para novos objetos criados no sistema. Estas definições são feitas no arquivo de configuração da seguinte forma:

```
type_transition type_1 type_2: file
      default_file_type;

type_transition type_1 type_2: process
      default_process_domain;
```

Já os papéis do RBAC são definidos como segue:

```
role rolename types { type_1 ... type_n };
```

Transições entre papéis podem ser definidas de forma a permitir mudanças de privilégio. Estas transições são definidas da seguinte forma:

```
role_transition current_role program_type new_role;
```

Estas construções no arquivo de configuração são utilizadas para definir o comportamento do TE e do RBAC do SELinux de acordo com as especificidades desejadas. Por exemplo, para utilização do domínio de administração do sistema, uma política poderia ser estabelecida para permitir que apenas a aplicação `login` tenha acesso a este domínio, impedindo acessos remotos de obterem os privilégios deste domínio. Tal política poderia ser implementada como segue:

```
type_transition getty_t login_exec_t:process
      local_login_t;
allow local_login_t sysadm_t:process transition;
allow newrole_t sysadm_t:process transition;
```

O processo de definição de políticas para o SELinux é complexo e resulta em um número muito alto de regras. Muitas distribuições GNU/Linux estabelecem as suas próprias políticas e as entregam prontas para uso aos seus usuários, como forma de facilitar o uso deste LSM. Outros sistemas MAC, por sua vez, tentam implementar geração automática de políticas e MACs simplificados, a saber LSMs TOMOYO e SMACK. Estes LSMs serão brevemente apresentados nas Seções 4.2.4 e 4.2.3, repectivamente. Ademais, pesquisadores buscam maneiras mais simples de definir políticas de segurança [38].

4.2.2 AppArmor

O AppArmor é um módulo de segurança Linux que implementa um MAC voltado para proteger e isolar processos. Este módulo LSM implementa o sistema de segurança SubDomain [14]. O AppArmor foi inicialmente desenvolvido pela empresa WireX, posteriormente passou a ser desenvolvido e mantido pela Novell e distribuído em sua distribuição SuSe Linux. Atualmente este projeto é mantido pela Canonical, mantenedora da distribuição Ubuntu Linux. A versão 2.4 do AppArmor, lançada por esta última empresa, foi aceita e incluída no Linux, a partir de sua versão 2.6.36.

O Subdomain foi concebido com a intenção de ter implementação e utilização simples. Características essas que, segundo os seus autores, são fundamentais para um sistema seguro, uma vez que implementações muito complexas de ferramentas de segurança estariam muito sujeitas a falhas de desenvolvimento que poderiam comprometer parcialmente ou totalmente a segurança do sistema. Configuração complexa também é um fator considerado crítico para sistemas de segurança, uma vez que pode induzir os administradores de sistemas a operar incorretamente as ferramentas adotadas, colocando assim a segurança do sistema em risco.

O AppArmor mantém essas características e utiliza os ganchos LSM para aplicar diferentes regras de acesso para diferentes perfis de processos. Estes perfis devem ser definidos em espaço de usuário e carregados para o módulo. Desta maneira, o AppArmor provê um mecanismo de isolamento de processos. Estas políticas não são globais, uma vez que devem ser definidas por processo e aqueles processos que não possuam políticas definidas não sofrem qualquer interferência do módulo, tendo seu controle de acesso regido apenas pelas funcionalidades DAC do Linux. Já os processos confinados, passam por checagens MAC definidas em seus perfis.

Tal isolamento tem como objetivo proteger o sistema operacional de ataques internos e externos, limitando o poder de ação de cada processo e forçando-os a manter um padrão de comportamento esperado. Desta forma, até mesmo falhas desconhecidas em aplicações (*zero-day*) podem, segundo os autores da ferramenta, ser prevenidas através de políticas bem definidas por processo, uma vez que, em caso de comprometimento, o processo atacado não terá grandes poderes de atuação sobre o sistema operacional. Esta ferramenta é tipicamente utilizada para isolar aplicações de serviços de rede, visando minimizar a possibilidade de ataque por comprometimento destes serviços.

Para facilitar ainda mais o seu uso, o AppArmor disponibiliza algumas políticas pré-definidas, que podem ser conciliadas com ajustes específicos do administrador. Outro recurso disponibilizado pela ferramenta é o modo de aprendizagem, onde aplicações são monitoradas segundo políticas definidas, e os casos de violação são registrados, sem serem evitados, para posterior análise e eventual ajuste de política por parte do administrador.

4.2.3 Smack

O Smack (*Simplified Mandatory Access Control Kernel*), como sugere a sua sigla, tem como objetivo a implementação de um sistema de controle de acesso mandatório simplificado.

O Smack adiciona restrições de acesso de sujeitos a objetos baseando-se em rótulos (*labels*) adicionados a ambas credenciais e regras de acesso globais (mandatórias). Rótulos são *strings* de um a 23 caracteres, sensíveis a caixa e não estruturados. As únicas operações que são feitas sobre os rótulos são operações de comparação, feitas durante a avaliação de regras de acesso. Alguns rótulos especiais são definidos pelo módulo LSM para estabelecer comportamentos específicos nas avaliações de acesso, a saber:

- `*` (*start*): utilizado para definir objetos que são acessados universalmente, contudo não devem compartilhar informações (ex. `/dev/null`). Sujeitos com este rótulo não podem acessar nenhum objeto do sistema. Objetos com este rótulo podem ser acessados por sujeitos com qualquer rótulo diferente deste;

- `_` (*floor*): utilizado para definir sujeitos de sistema. Sujeitos com este rótulo possuem acesso de leitura para objetos com rótulo *floor*;

- `^` (*hat*): utilizado para definir sujeitos com acesso de leitura global. Sujeitos com este rótulo possuem acesso de leitura a qualquer objeto do sistema.

As regras de acesso (*rules*) consideram os mesmos tipos de acesso padrão do Linux: leitura, escrita e execução. Acessos do tipo *append* (concatenação) e *transmute* também são considerados nas regras do Smack. Estas regras são compostas na forma "`subject-label object-label access`", onde *access* admite os valores `r`, `w`, `x`, `a` e `t` para representar respectivamente o modos de acesso anteriormente descritos. Este último modo de acesso, *transmute* representa permissão para um objeto do tipo diretório forçar que o rótulo de um objeto criado dentro de sua hierarquia seja igual ao seu, e não igual ao do sujeitos que o criou. Para os casos em que nenhum acesso deve ser permitido, o caractere "`-`" deve ser utilizado sozinho no campo `access` da regra.

Alguns exemplos de regras válidas são:

```
TopSecret Secret  rx
Secret    Unclass R
Manager   Game    x
User      HR      w
New       Old     rRrRr
Closed    Off     -
```

Estas regras regras são definidas no arquivo `/etc/smack/accesses`, de forma a serem carregadas na inicialização do sistema operacional e podem, a qualquer tempo serem definidas através do arquivo `/smack/load` disponibilizado pela *interface SecurityFS* do LSM Smack.

Todos os processos do sistema operacional recebem rótulos. Os rótulos de processos do sistema são definidos como "_" automaticamente pelo Smack em tempo de *boot*. Rótulos de processos de usuário são definidos de acordo com as regras estabelecidas no arquivo de configuração `/etc/smack/user`.

Os rótulos de objetos são definidos através do atributo estendido de segurança (*xattr security namespace*, Seção 3.5) SMACK64.

As regras de acesso são avaliadas pelo Smack da seguinte forma:

1. Acessos de qualquer natureza feitos por sujeitos com rótulo "*" são negados;
2. Acessos de leitura e execução feitos por sujeitos com rótulo "^" são autorizados;
3. Acessos de leitura e execução feitos sobre objetos com rótulo "_" são autorizados;
4. Acessos de qualquer natureza feitos sobre objetos com rótulo "*" são autorizados;
5. Acessos de qualquer natureza feitos por sujeitos de rótulo igual ao do objeto sendo acesso são autorizados;
6. Acessos cujo perfil corresponda a alguma regra explicitamente definida no conjunto de regras carregado são autorizados;
7. Qualquer outro tipo de acesso é negado.

Sistemas MAC baseados em comportamento de aplicações, tais como SELinux e AppArmor, devem ser customizados para cada distribuição GNU/Linux em quem são utilizados. Desenvolvedores de sistemas embarcados, por sua vez, não costumam desejar as integrações oferecidas por tais distribuições. Dada a sua simplicidade e independência de aplicações específicas, o Smack tem sido muito utilizado em sistemas com Linux embarcado [57].

4.2.4 TOMOYO

O TOMOYO Linux é uma implementação de controle de acesso mandatório (MAC) para *kernel* Linux. Este módulo LSM difere dos apresentados anteriormente pois seu critério de avaliação não é baseado em permissões de acesso a objetos (*inode based security)*, mas sim em controle de acesso baseado em domínios (*name based security*).

Conforme ilustrado no funcionamento das soluções apresentadas até o momento, o comportamento de soluções MAC é totalmente controlado pelas suas políticas. Diante da complexidade observada na elaboração de políticas de segurança para alguns sistemas de controle de acesso mandatório, os desenvolvedores do TOMOYO constataram que a geração de políticas é a chave para melhorar a segurança de sistemas Linux [37]. A partir desta constatação, estes desenvolvedores iniciaram pesquisas por uma solução de geração automática de políticas, baseando-se na observação da execução de processos [38].

O objetivo original do projeto era a geração automática de políticas para o SELinux, o que não foi possível devido à falta de hierarquia na definição dos domínios de segurança daquele LSM. A evolução da pesquisa, contudo, resultou na criação do LSM TOMOYO, um sistema MAC que implementa uma metodologia de criação automática de políticas de segurança baseada na observação de mudanças de estados de processo. Esta observação é feita basicamente através da observação da sequência histórica de comandos (*path names*) invocados por processos.

O TOMOYO introduz os seguintes conceitos nas suas definições de políticas: cada processo faz parte de um único domínio. A criação de domínios e o trânsito entre eles acontece sempre que um novo processo é executado. A divisão de domínios é feita automaticamente em nível de *kernel*. Cada domínio possui suas restrições de acesso a objetos com granularidade do tipo `rwx` (leitura, escrita e execução). As políticas MAC são criadas automaticamente a partir da observação do comportamento dos domínios.

Permissões são atribuídas a domínios, e não a objetos ou sujeitos. Por este motivo, não existe o conceito de *object owner* (UID). O domínio inicial é `<kernel>`. Os domínios subsequentes são definidos através da concatenação do histórico de comandos invocados por processos. A Figura 4.3 ilustra como os domínios deste módulo LSM são estabelecidos e seus respectivos nomes.

As regras MAC do TOMOYO atuam sobre arquivos, conexões de rede (em nível de porta e protocolo[11]), e POSIX *Capabilities*. É possível a definição de *trusted domains*, que são domínios onde apenas as checagens DAC serão feitas.

4.2.5 Yama

Adicionado no Linux 3.4, o Yama é um módulo LSM que tem como objetivo aumentar a granularidade de controle de acesso à chamada de sistema `ptrace()`.

A chamada de sistema `ptrace()` permite a um processo observar e controlar a execução de processos com UID igual ao seu, podendo examinar e alterar seus comportamentos. Os principais objetivos desta chamada são monitoramento de chamadas de sistema e implementação de *breakpoints* para depuração de código [1]. Esta característica permite a

[11] Layer 3 - TCP ou UDP

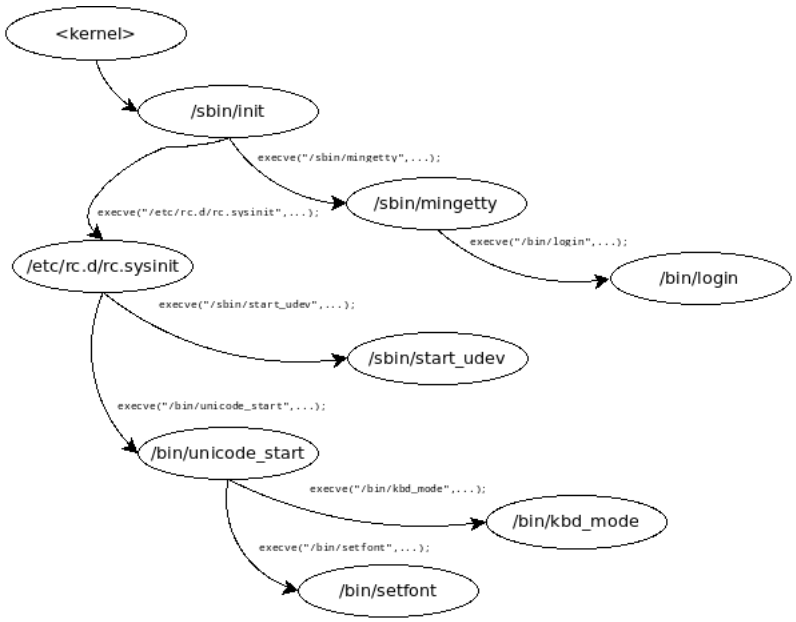


Figura 4.3: TOMOYO: geração automática de domínios de segurança baseada em nomes.

um atacante com um processo comprometido, utilizar esta chamada para continuar com o seu ataque. Exemplos de ataques com este perfil são: roubo de sessões SSH e ataques de injeção de código [12].

Por este motivo, o Yama implementa uma lógica de maior granularidade para controle de acesso a esta chamada. Este LSM define níveis, onde o comportamento do controle de acesso à chamada possui diferentes características, a saber:

- Modo 0: permissões tradicionais da chamada `ptrace()`. Um processo pode fazer a chamada `ptrace(PTRACE_ATTACH, pid, ...)` desde que possua o possua o mesmo UID do processo indicado pela variável `pid` e este processo tenha habilitado operações de depuração (`prctl(PR_SET_DUMPABLE, ...)`);

- Modo 1: acesso restrito à chamada `ptrace()`. Um processo pode fazer a chamada `ptrace(PTRACE_ATTACH, pid, ...)` desde que possua o mesmo UID do processo indicado pela variável `pid`, este processo tenha habilitado operações de depuração (`prctl(PR_SET_DUMPABLE, ...)`) e o processo a ser monitorado seja seu filho;

- Modo 2: somente administradores. Apenas processos com CAP_SYS_PTRACE podem fazer chamadas `ptrace(PTRACE_ATTACH, ...)`;

- Modo 3: acesso bloqueado. Nenhum processo pode fazer chamadas `ptrace()` do tipo `PTRACE_ATTACH`. Uma vez definido, este nível não pode ser mudado nem mesmo pelos administradores do sistema.

A manipulação dos níveis de controle de acesso é feita através de um arquivo disponibilizado na *interface SecurityFS* do Yama, no qual pode-se ler o estado atual do sistema ou pode-se definir um novo estado. Apenas processos com CAP_SYS_PTRACE podem alterar este arquivo de controle:

`/proc/sys/kernel/yama/ptrace_scope`

4.2.6 Outras soluções

Vários outros projetos de pesquisa em segurança buscam o desenvolvimento de soluções que elevem o nível de segurança do Linux através de diferentes metodologias de controle de acesso. Muitos não utilizam o *framework* LSM, por considerarem que este representa um limitador para a implementação de soluções realmente seguras [54] [5].

O projeto GRSecurity [32] trabalha no desenvolvimento de um sistema de segurança de *kernel* capaz de oferecer proteção contra ataques desconhecidos (*zero day*) através de melhorias no sistema de autenticação, auditoria em importantes eventos do sistema operacional e controle de acesso MAC do tipo RBAC. Este projeto não utiliza LSM em sua implementação.

O RSBAC [53] implementa um *framework* de *kernel* que permite a utilização de diferentes modelos de segurança a partir módulos de decisão dinamicamente carregáveis. Estes módulos podem ser: MAC, ACL, Anti-vírus, dentre outros. Este trabalho também não utiliza LSM em sua implementação.

Hallyn et al [36] desenvolveu um mecanismo de controle de acesso chamado *Domain Type Enforcement* (DTE) para Linux. Esta solução é uma variação do *Type Enforcement* que utiliza mecanismos baseados em nomes de arquivos (*pathnames*).

Outras soluções de segurança, por sua vez, utilizam o *framework* LSM para implementação de soluções não diretamente relacionadas a controle de acesso. Os trabalhos do projeto Imuno, descritos na Seção 2.2, por exemplo, visam a construção de sistema de segurança imunológico geral, implementado em ambiente Linux. O trabalho proposto nesta tese de mestrado é parte integrante desta pesquisa e utiliza o *framework* LSM na implementação do sistema proposto. Já Takamasa utiliza o LSM no desenvolvimento de um sistema de monitoramento de segurança [41].

4.3 Security FS

O *Security FS* é um sistema de arquivos virtual criado com o propósito de ser API comum para a criação de *interfaces* de módulos de segurança que utilizem o *framework* LSM. Motivado pela constatação de que vários módulos LSM estavam implementando

suas próprias *interfaces* [13], o *Security FS* procura minimizar o impacto no código do *kernel* gerado por esta prática, além de simplificar a manutenção de módulos LSM.

Neste sistema de arquivos, todos os arquivos e diretórios são definidos (criados) por um módulo LSM. Uma aplicação em espaço de usuário poderá fazer operações de leitura e escrita sobre os arquivos disponibilizados. Não poderá, contudo, criar ou remover arquivos e diretórios, tampouco mudar permissões de leitura e escrita dos arquivos existentes.

As operações de leitura e escrita em espaço de usuário sobre os arquivos do *Security FS* serão tratadas através de *callbacks* pelo módulo LSM, o qual receberá uma cópia dos parâmetros das chamadas de sistema das respectivas operações e atuará de maneira adequada, ajustando o seu comportamento a partir de parâmetros recebidos do espaço de usuários (operações de escrita) ou enviando informações de espaço de *kernel* para o espaço de usuário (operações de leitura).

O sistema de arquivos é montado automaticamente em `/sys/kernel/security`.

A API do sistema de arquivo exporta três funções básicas: `securityfs_create_dir()`, `securityfs_create_file` e `securityfs_remove()`. Estas funções estão definidas em `linux-src/security/inode.c` e possuem as seguintes assinaturas:

```
extern struct dentry *securityfs_create_file(
                                const char *name, umode_t mode,
                                struct dentry *parent,
                                void *data,
                                const struct file_operations *fops);

extern struct dentry *securityfs_create_dir(
                                const char *name,
                                struct dentry *parent);

extern void securityfs_remove(struct dentry *dentry);
```

Para utilizar este sistema de arquivos, um módulo LSM deve primeiramente incluir o cabeçalho `<linux/security.h>`. Em seguida, o módulo deve definir as *callbacks* que farão o tratamento das chamadas do sistema de arquivos virtual. Esta definição é feita através da declaração de uma estrutura do tipo `file_operations`. Um possível exemplo para um módulo LSM chamado `mod` seria:

```
static const struct file_operations mod_operations = {
        .read    = mod_callback_for_read,
        .write   = mod_callback_for_write,
```

```
        .open    = mod_callback_for_open,
        .release = mod_callback_for_release,
};
```

Por fim, o módulo LSM deve criar uma função de inicialização do sistema de arquivos, onde devem ser definidos os diretórios e arquivos que o compõe, e, posteriormente, fazer uma chamada para a macro `fs_initcall()` para inicializar o sistema.

Continuando o exemplo anterior, possíveis implementações da função de iniciação e da *callback* `open()` para o módulo `mod` são apresentadas abaixo.

- Inicialização do sistema de arquivos *Security FS*, feita através da chamada `fs_initcall(mod_vfs_init)`. Esta função cria um diretório `mod` na raiz do sistema de arquivos e, em seguida, um arquivo chamado `readme` no diretório recém criado:

```
int mod_vfs_init(void)
{
        struct dentry *op_dir;
        op_dir = securityfs_create_dir("mod", NULL);
        securityfs_create_file("readme", 0400, op_dir,
                               NULL, &mod_operations);
return 0;
}
```

- *Callback* para tratamento de chamadas de sistema `open()` sobre arquivos do *Security FS*. Esta função verifica o nome do arquivo lido e, caso seja `readme`, cria em memória um arquivo com a *string* "*This is a LSM Module*", a qual posteriormente será enviada para espaço de usuário através da chamada de sistema `read()`:

```
int mod_callback_for_open(struct inode *inode, struct file *file)
{
        struct qstr *file_name;
        struct mod_data *data;
        char buffer[200] = "This is a LSM Module";
        file_name = &(file->f_dentry->d_name);
        if (strcmp(file_name->name, "readme") == 0) {
                data = kmalloc(sizeof(*data), GFP_KERNEL);
                if (data){
                        data->read_eof = false;
                        data->buffer = kmalloc(IO_BUFFER_SIZE,
```

```
                                                    GFP_KERNEL);
                            memcpy(data->buffer, buffer, len);
                    }
            file->private_data = data;
            }
            return 0;
    }
```

4.4 Netfilter

O Netfilter é um *framework* do *kernel* Linux que adiciona ganchos em pontos do seu código onde é possível interceptar o fluxo de pacotes de rede.

Os ganchos Netfilter estão dispostos em diferentes pontos do fluxo de pacotes de rede do *kernel*, conforme ilustra a Figura 4.4. O Netfilter permite o registro de várias funções tratadoras para cada um desses ganchos.

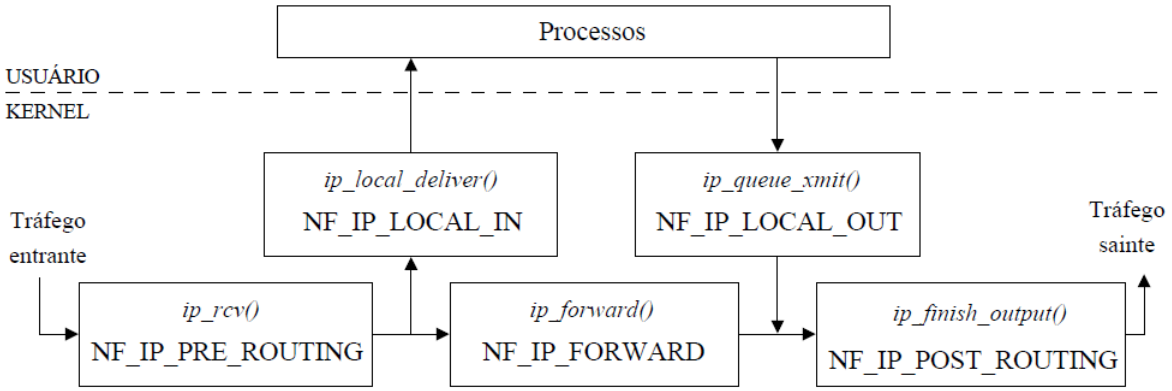


Figura 4.4: Arquitetura de ganchos do *framework* Netfilter.

Quando alcançados, os ganchos Netfilter desviam para a um função controladora de gancho, `NF_HOOK()`, a qual aciona sequencialmente os tratadores registrados para aquele determinado gancho, enviando um cópia do pacote capturado. Cada função tratadora deve então analisar o conteúdo do pacote enviado e retornar um dos valores abaixo:

- `NF_DROP`
- `NF_ACCEPT`
- `NF_STOLEN`
- `NF_QUEUE`

- NF_REPEAT
- NF_STOP

O controlador de ganchos aciona a próxima função tratadora de um determinado gancho sempre que recebe o valor NF_ACCEPT como retorno da última tratadora executada. Caso o valor de retorno seja NF_DROP, o pacote é imediatamente rejeitado pelo Netfilter. Caso o valor seja NF_STOP, o pacote é imediatamente aceito sem que outras tratadoras sejam invocadas. Uma função tratadora pode ignorar um determinado pacote sem aceitá-lo ou rejeitá-lo retornado o valor NF_STOLEN, que fará com que o controlador desconsidere aquele tratador e acione o próximo na lista. Um tratador pode ainda solicitar que seja reinvocado, retornando o valor NF_REPEAT. Por fim, um pacote pode ser enviado para análise em espaço de usuário sempre que um tratador retornar o valor NF_QUEUE. Os pacotes enviados para espaço de usuário são recebidos e tratados por aplicações através do uso da biblioteca libiqp. A aplicação em espaço de usuário responsável pela avaliação do pacote deve retornar sua decisão sobre o pacote para o gerenciador de ganchos do Netfilter.

4.5 TCG - Task Control Group

O TCG *Task Control Group*, é um *framework* do Linux, inserido a partir de sua versão 2.6.24[47], com o objetivo de disponibilizar funcionalidades de separação e organização de processos em grupos, onde, para cada grupo, fosse possível definir regras de comportamento aos processos que o compõe. Desta forma, o TCG disponibiliza uma API para agrupamento e monitoramento de processos do sistema operacional. Esta API é requisito comum para mecanismos de controle de recursos do sistema operacional, tais como mecanismos para controle de consumo de CPU, uso de banda de rede, uso de memória e outros, de tal forma que os desenvolvedores destes mecanismos não precisam se preocupar em como agrupar e monitorar processos, tampouco em ter de desenvolver uma *interface* de usuário especifica do mecanismo. Desta forma, o TCG simplifica a implementação de mecanismos de controle de recursos, chamados de subsistemas, e permite que estes representem um menor impacto no código do *kernel* uma vez que evita duplicação de rotinas e estruturas de dados.

4.5.1 Arquitetura e implementação

Para atender as necessidades comuns de diferentes subsistemas, o *framework* TCG foi projetado para atender os seguintes requisitos[48]:

1. Múltiplos subsistemas independentes: como os subsistemas do TCG são tipicamente ferramentas de contabilização e controle do uso de diferentes recursos, deve ser possível o funcionamento concorrente de diferentes subsistemas. Deve ser possível também ao usuário escolher quais dos subsistemas disponíveis serão utilizados em um dado momento

2. Mobilidade: um usuário com permissões adequadas deve ter o poder de mover os processos entre grupos

3. Inescapabilidade: uma vez estabelecidos os grupos aos quais cada processo pertence, um processo não deve ser capaz de mudar de grupo sem a intervenção de um usuário administrador que possua o nível de permissão adequado para a tarefa

4. Interface extensível: diferentes subsistemas possuem diferentes parâmetros a serem configurados. Os usuários, contudo, esperam padronização na utilização destes subsistemas. O TCG deve, portanto, fornecer uma *interface* extensível e com algum nível de padronização. Neste contexto, duas possíveis soluções seriam: nova chamada de sistema (*syscall*) ou um sistema de arquivos virtual, em que os subsistemas disponibilizassem arquivos virtuais onde os parâmetros de configuração pudessem ser ajustados diretamente através de operações de leitura e escrita, em espaço de usuário, sobre eles. A primeira alternativa oferece um alto grau de padronização e bom desempenho. Esta abordagem, contudo, pode ser muito restritiva para os subsistemas, devido a variabilidade de seus conjuntos de parâmetros de configuração. A segunda opção, por sua vez, apesar de possuir um desempenho menor que a primeira, provê grande flexibilidade aos subsistemas e tem a vantagem de ser facilmente manipulável através de ferramentas padrão e rotinas de biblioteca em espaço de usuário. O TCG foi implementado utilizando esta última abordagem

5. Níveis estruturados de grupos: para alguns subsistemas, pode ser interessante a habilidade de estruturar os grupos de processos em diferentes níveis hierárquicos. Nesta estrutura, é possível que uma parcela dos recursos alocados a um grupo de processos seja reservada a um sub-grupo. A Figura 4.5 ilustra uma possível utilização desta funcionalidade por um eventual subsistema que implemente controle do uso de banda de rede.

6. Múltiplas partições: considerando o Requisito 1, independência entre subsistemas, a definição de uma única divisão hierárquica dos grupos de processos apresentada no requisito anterior penaliza a independência entre os subsistemas, uma vez que diferentes subsistemas precisariam compartilhar o mesmo critério de divisão de processos. Por isso mesmo, o TCG deve permitir a criação de diferentes hierarquias,

de tal forma que seja possível estabelecer diferentes critérios de divisão para cada subsistema. A Figura 4.6 ilustra uma possível utilização de múltiplas partições. É desejável também que, para os casos onde dois ou mais subsistemas compartilhem o mesmo critério de divisão de grupos, a definição de hierarquias seja feita de maneira simplificada

7. Associação de objetos genéricos a grupos: para alguns subsistemas, somente objetos do tipo processo podem não ser suficientes para a implementação de monitoramento e controle sobre o recurso desejado. Este requisito permite que um subsistema associe a ele outros tipos de objetos, tais como páginas de memória, descritores de arquivos e *sockets* de rede

8. Desempenho: por fim, a implementação deve ser feita de maneira tal que sobrecarregue minimamente o processamento.

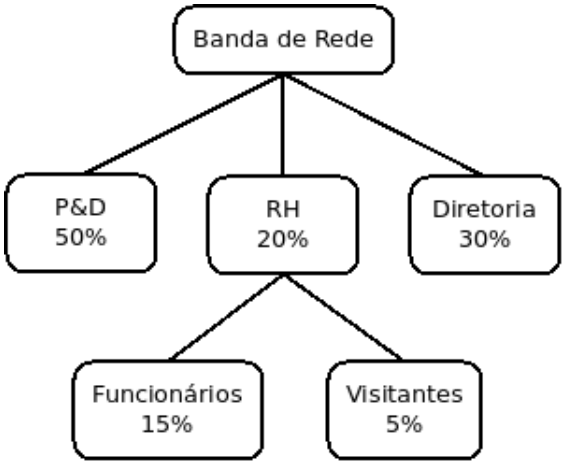


Figura 4.5: Exemplo de distribuição de banda de rede baseado em grupos.

O TCG é implementado através da inclusão de ganchos em pontos do *kernel* relacionados à criação e finalização de processos (`fork()` e `exit()`). Sempre que esses ganchos são atingidos, o controle é desviado para as funções do TCG que implementam a lógica de agrupamento de processos tal como descrito nos requisitos apresentados. Estas funções, atualizam as estruturas que compõem a arquitetura do TCG, relacionadas abaixo e, quando aplicável, deviam o fluxo de execução para os respectivos subsistemas atuarem sobre os processos. O relacionamento entre estas estruturas é apresentado na Figura 4.7.

- `css_set`: representa o conjunto de processos monitorados por um determinado grupo. Nesta estrutura são armazenados o grau de parentesco entre os grupos da hierarquia, *flags* de estado do grupo e ponteiros para o conjunto de

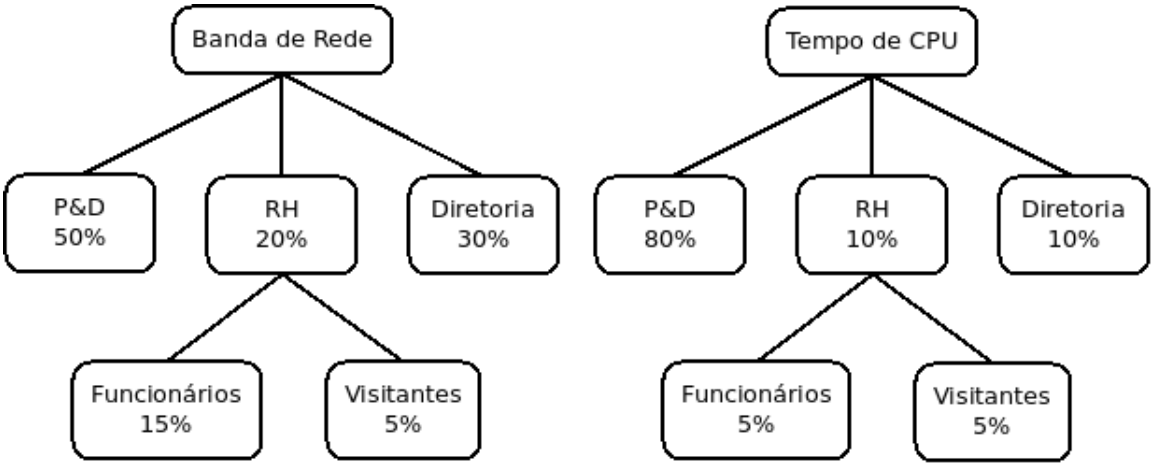


Figura 4.6: Exemplo de distribuição de recursos com diferentes critérios por recurso.

`container_subsys_state`, sendo um para cada subsistema utilizado. Nesta estrutura não são mantidas informações sobre a lista de processos que fazem parte do grupo, uma vez que operações que acessam os dados armazenados em `css_set` ocorrem frequentemente, equanto operações que acessam informações relativas a listagem de processos do grupo, tal como movimentação de processos entre grupos, acontecem com menor frequência. É possível, contudo, remontar a lista de processos do grupo seguindo o ponteiro para lista ligada `tasks` existente na estrutura.

- `cgroup_subsys`: representa um subsistema. Nesta estruturas ficam armazenadas as *callbacks* para os ganchos implementados pelo *framework*. Através delas, os subsistemas podem colher informações sobre os processos e interferir no seu comportamento. São elas:
  - `create`: chamada quando um novo grupo (`css_set`) é criado
  - `destroy`: chamada quando um novo grupo é removido
  - `can_attach`: chamada imediatamente antes da inclusão de um processo em um grupo. Esta *callback* permite que o subsistema negue a inclusão
  - `attach`: chamada quando um processo é incluído em um grupo
  - `fork`: chamada quando um novo processo é criado em um grupo através da chamada de sistema *fork()*
  - `exit`: chamada quando um processo é finalizado (chamada de sistema *exit()*)
  - `populate`: chamada quando um novo grupo é criado. Esta *callback* deve incluir os arquivos virtuais de controle do subsistema no diretório refente a este novo grupo

  - `bind`: chamada quando um subsistema é movido entre diferentes hierarquias

- `cgroup_subsys_state`: estrutura para armazenamento de informações de estado por dupla grupo/subsistema. As principais informações armazenadas nessa estrutura são:

  - `cgroup`: ponteiro para o grupo ao qual este subsistema está anexado. Importante para a obtenção de informações sobre estrutura hierarquia
  - `refctn`: observando a Requisito 7 apresentado anteriormente, esta variável é um contador de referências a objetos que não sejam do tipo processo pertencentes ao subsistema. Esta variável garante que não seja possível a remoção de um grupo que tenha objetos genéricos associados, mesmo quando não houver mais nenhum processo associado a ele

- `task_group`: armazena ponteiros para cada subsistema registrado e outras variáveis de controle de comportamento específicas de cada subsistema. Manter ponteiros referentes ao controle de grupos (hierarquias, subsistemas, etc) no contexto de um processo foi considerado um desperdício pelos desenvolvedores do TCG, principalmente para os casos onde subsistemas estiverem compilados no *kernel*, sem contudo serem utilizados. Além disso, diversos processos membros de um mesmo grupo, compartilhariam as mesmas informações de subsistemas e hierarquias, o que reforçaria o desperdício. Desta forma, com a definição desta estrutura, os descritores de processo `task_struct` podem ser simplesmente acrescidos de um ponteiro para a estrutura `task_group` do grupo ao qual o processo pertence

Na Figura 4.7, tem-se um *kernel* compilado com três subsistemas disponíveis: `A`, `B` e `C`. O administrador do sistema monta uma única hierarquia com os subsistemas `A` e `B` ligados a ela. Dois grupos, `grp1` e `grp2`, são criados. Os processos `T1` e `T2` são alocados em `grp1` e o processo `T3` é alocado em `grp2`. Tem-se então uma estrutura `cgroup_subsys_state` para cada par grupo/subsistema e uma estrutura `task_group` para cada grupo.

O registro de subsistemas é feito em tempo de compilação do *kernel*. Por essa razão, não é possível criar um subsistema na forma de LKM.

O TCG não é classificado exatamente como um *framework* de segurança do Linux, contudo, será visto no Capítulo 5 que a capacidade de controlar a oferta e uso de recursos do sistema operacional por processos é um requisito de grande importância na implementação de mecanismos de resposta a ataques em um sistema de segurança imunológico.

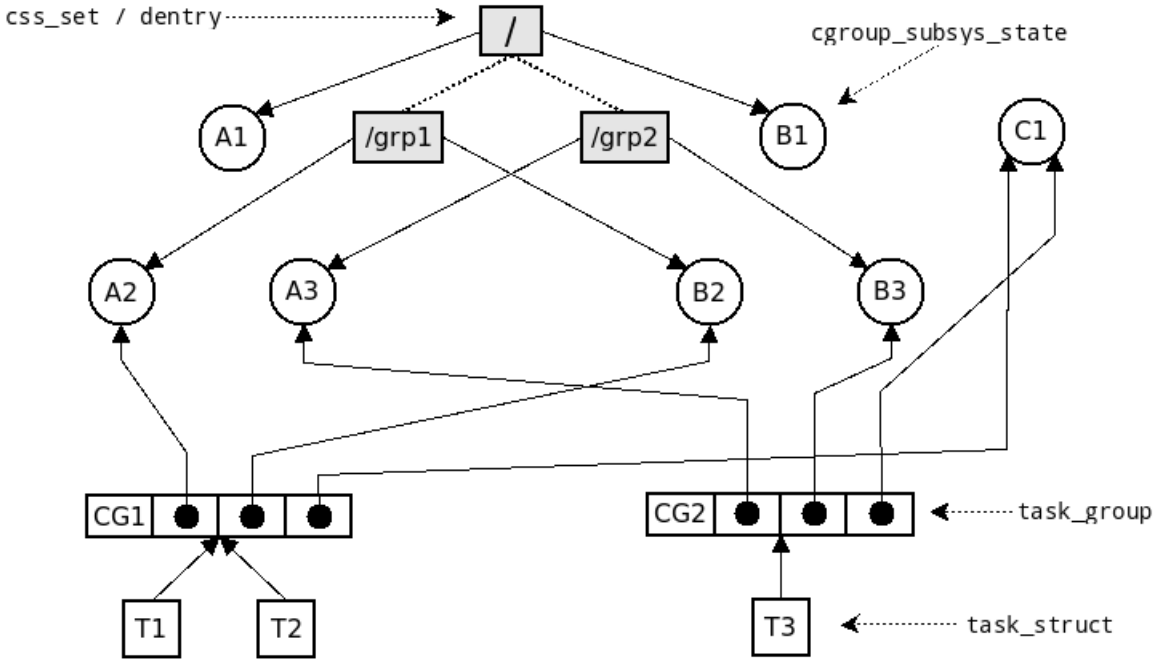


Figura 4.7: Relacionamento entre as estruturas do TCG. Adaptado de [48].

4.5.2 Funcionamento

O TCG faz *interface* com o espaço de usuário via sistema de arquivos virtual próprio, do tipo `cgroup`, através do qual é possível criar e manter hierarquias, movimentar processos entre grupos, além de ajustar os parâmetros oferecidos pelos subsistemas.

Uma hierarquia estruturada em forma de árvore, onde todos os processos do sistema operacional encontram-se exclusivamente em um grupo. Estas hierarquias possuem subsistemas associados, atuando sobre todos os seus grupos. É possível a existência de mais de uma hierarquia simultaneamente no sistema, cada uma com a sua própria estrutura de divisão de grupos.

A criação de uma hierarquia é feita através da utilização da chamada de sistema `mount()` para montagem do sistema de arquivos virtual do tipo `cgroup`. Através das opções de montagem desta chamada, pode-se definir qual ou quais dos subsistemas disponíveis será utilizado naquela hierarquia que está sendo criada/montada. Caso tente-se utilizar um subsistema que já esteja em uso por outra hierarquia, a operação falha e o valor `EBUSY` é retornado. Por outro lado, caso a hierarquia que está sendo montada seja exatamente igual a outra existente, ou seja, utilize exatamente os mesmos subsistemas, o mesmo superbloco é utilizado.

Após a criação de uma hierarquia, todos os processos existentes no sistema operacional ficam alocados no grupo raiz. A construção da estrutura da hierarquia é feita através da

criação de diretórios, via chamada de sistema `mkdir()`. Dentro do diretório de cada grupo, existe o aquivo de controle `tasks`, onde estão listados todos os processos que fazem parte daquele grupo. A leitura deste arquivo retorna a lista dos processos que fazem parte daquele grupo, ao passo que a escrita neste arquivo move o processo entre grupos. Outros arquivos de controle podem ser incluídos nos diretórios do grupo pelos subsistemas.

A sequência de comandos abaixo listada ilustra a criação da hierarquia apresentada na Figura 4.7[12]:

```
# mkdir /tcg
# mount -t cgroup -o A,B none /tcg
# mkdir /tcg/grp1 /tcg/grp2
# echo 1011 > /tcg/grp1/tasks
# echo 1012 > /tcg/grp1/tasks
# echo 1013 > /tcg/grp2/tasks
```

O TCG disponibiliza ainda outros dois aquivos de controle: `/proc/cgroups`, que contém informações sobre o uso dos subsistemas disponíveis (compilados) no *kernel* e `/proc/<PID>/cgroup`, que lista o caminho dos grupos aos quais o processo faz parte em cada hierarquia ativa.

Quando uma hierarquia é desmontada, é necessário que não haja nenhum subgrupo existente para que a hierarquia seja extinta e os seus subsistemas liberados para utilização por outras hierarquias. Caso existam subgrupos, o sistema de arquivos virtual da hierarquia é desmontado, porém a hierarquia continua existindo, assim como o seu superbloco, apesar e não estar visivelmente montada no sistema de arquivos. A remoção de grupos é feita através da chamada `rmdir()` e só pode ser feita caso não haja nenhum processo atrelado ao grupo que está sendo removido.

4.5.3 Subsistemas

Abaixo são apresentados brevemente os subsistemas disponíveis na versão 3.7.6 do Linux.

cpusets

O subsistema *cpusets* pode ser considerado como o primeiro subsistema TCG a fazer parte da versão oficial do Linux. Este subsistema, na verdade, já estava disponível no *kernel* antes mesmo do TCG e foi um dos motivadores para a criação do *framework* [48]. Com a disponibilidade do TCG, a partir do Linux 3.6.24, o *cpusets* foi remodelado

[12]Neste exemplo considera-se que os PIDs de T1, T2 e T3 são, respectivamente, 1011, 1012 e 1013.

como um subsistema TCG, o que resultou em uma redução significativa no tamanho do seu código, devido à eliminação dos trechos responsáveis por gerenciamento de grupo de processos e tratamento de *interface*. Este subsistema é compilado no *kernel* habilitando-se a variável `CONFIG_CPUSETS` e é atribuído a uma hierarquia através da opção `cpuset` da chamada de sistema `mount()`.

Este subsistema provê mecanismos para atribuição de subconjuntos de CPUs e nós de memória exclusivamente para um conjunto de processos, além de oferecer outros recursos avançados no uso de CPU e memória.

O subsistema *cpusets* disponibiliza os seguintes arquivos de controle:

- `cpuset.cpus`: CPUs válidos para o grupo
- `cpuset.mems`: nós de memória válidos para o grupo
- `cpuset.cpu_exclusive`: *flag* de exclusividade. Caso um grupo tenha esse parâmetro habilitado, nenhum outro grupo além dos seus pais ou filhos podem compartilhar uma CPU
- `cpuset.mem_exclusive`: *flag* de exclusividade. Caso um grupo tenha esse parâmetro habilitado, nenhum outro grupo além dos seus pais ou filhos podem compartilhar nós de memória
- `cpuset.mem_hardwall`: restringe alocações de *kernel* para páginas, *buffers* e outros dados que são compartilhados entre aplicações em espaço de usuário
- `cpuset.memory_migrate`: uma vez que uma página é alocada, esta página permanece no mesmo nó enquanto ela permanecer alocada, mesmo que a política de gerenciamento de memória (`cpuset.mems`) do seu grupo mude. Se este *flag* estiver habilitado, contudo, as páginas alocadas a um processo do grupo serão migradas para os nós dentro da nova política do grupo
- `cpuset.memory_pressure`: métrica da taxa com que processos estão tentando liberar memória utilizada para atender a novas requisições de uso de memória. É um número inteiro que representa a taxa recente de requisições de página por processo do grupo em número de requisições por segundo vezes mil
- `cpuset.memory_spread_page`: *flag* booleana que controla a localização da alocação de memória para *buffers* de sistemas de arquivos e estruturas de dados de *kernel* relacionadas. Desabilitado por padrão. Caso habilitado, permite que tais espaços possam ser alocados em qualquer nó de memória ao qual o processo possua acesso, e não exatamente no nó onde o processo está rodando

- `cpuset.memory_spread_slab`: *flag* booleana que controla a localização da alocação memória para estruturas de dados de *kernel*. Desabilitado por padrão. Caso habilitado, permite que tais espaços possam ser alocados em qualquer nó de memória ao qual o processo possua acesso, e não exatamente no nó onde o processo está rodando
- `cpuset.sched_load_balance`: quando habilitada (configuração padrão), esta *flag* garante o balanceamento de carga entre as CPUs do grupo
- `cpuset.sched_relax_domain_level`: controla o comportamento de busca por CPUs ociosas no grupo

Group CPU Scheduler

Introduzido a partir da versão 2.6.24, o subsistema *Group CPU Scheduler* é compilado no *kernel* habilitando-se a variável `CONFIG_CGROUP_SCHED` e é montado através da opção `cpu` da chamada de sistema `mount()`.

Este subsistema tem como objetivo implementar o mecanismo de escalonamento baseado em grupos do CFS.

O escalonador CFS, introduzido no *kernel* 2.6.23, foi desenvolvido com o objetivo de entregar uma distribuição justa de tempo de CPU[63]. No seu comportamento padrão, este escalonador opera sobre processos individuais do sistema operacional e tenta prover tempos de CPU iguais para eles.

Contudo, algumas vezes deseja-se agrupar conjuntos de processos e prover tempo de CPU justo para cada processo do grupo. Por exemplo, pode-se desejar primeiro prover tempo de CPU justo para cada usuário do sistema e, então, tempo justo para cada processo pertencente a um usuário. Este desejo parte da necessidade de evitar que um usuário possa criar muitos processos e, consequentemente, receber mais tempo de CPU que um outro usuário (com menos processos). Por exemplo: quando tem-se quatro usuários rodando um processo cada um, o escalonador tende a oferecer 25% de tempo de CPU para cada um deles, o que é justo. Contudo, caso um destes usuários (usuário A) crie mais 96 processos, o escalonador passará a oferecer 1% de CPU para cada um dos 100 processos, ou seja o usuário A usará 97% de CPU, enquanto os outros três usuários usarão apenas 1% de CPU cada um, o que pode ser considerado injusto.

Por este motivo, o CFS introduziu o conceito de *Group Scheduling* (`CONFIG_GROUP_SCHED`), onde o escalonador pode ser ajustado de forma a entregar tempos de CPU iguais para grupos de processo, independente de quantos processos fazem parte de cada grupo. Esta abordagem elimina o problema de justiça de escalonamento apresentado no parágrafo anterior.

O CFS permite dois mecanismos mutuamente exclusivos de agrupamento de processos: baseado em usuários (`CONFIG_USER_SCHED`) e baseado em grupos (`CONFIG_CGROUP_SCHED`).

No primeiro caso, os processos são agrupados por usuário, de acordo com o seu `UID` (*User IDentification*). Um diretório é criado no sistema de arquivos virtual `sysfs` para cada usuário (`/sys/kernel/uids/<UID>`) e um arquivo de controle `cpu_share` é adicionado nestes diretórios. No segundo caso, os processos são agrupados usando a infraestrutura do TCG, onde cada `cgroup` possui um arquivo de controle `cpu_share`.

Em ambos os casos, a razão entre os valores dos arquivos `cpu_share` dos diferentes usuários/grupos define a razão do tempo de CPU de cada grupo.

CFS Bandwidth Control

Introduzido a partir da versão 3.2 do Linux, o *CFS Bandwidth Control* é uma extensão do subsistema *Group CPU Scheduler* e é compilado no *kernel* habilitando-se a variável `CONFIG_CFS_BANDWIDTH`. Este subsistema também é atribuído a uma hierarquia através da opção `cpu` da chamada de sistema `mount`.

Um escalonador de processos em um sistema operacional divide o tempo de CPU entre os processos que desejam ser executados (multiprocessamento). Não existem limites superiores para o uso de CPU por um processo quando não existem outros processos competido por este recurso. Em algumas situações, contudo, pode ser desejável limitar o uso de CPU por um dado processo, ou grupo de processos, mesmo que não existam outros processos tentando ser executados. Por este motivo, o *CFS Bandwidth Control* implementa um mecanismo para limitação do tempo de uso de CPU por parte de um grupo de processos [62].

A definição deste limite é feita através da definição de um período de tempo de processamento e, posteriormente, configuração de uma quota de tempo a ser utilizada pelo grupo dentro de cada período determinado.

O subsistema utiliza os seguintes arquivos de controle:

- `cpu.cfs_period`: tempo, em microsegundos, da unidade de referência (período). O valor máximo desta variável é $1000000\mu s$ (um segundo)

- `cpu.cfs_quota`: tempo máximo, em microsegundos, de utilização de CPU para cada período. Caso essa variável possua valor negativo, não serão impostos limites. O mínimo valor possível desta variável é $1000\mu s$ (um milisegundo).

Simple CPU Accounting

Introduzido a partir da versão 2.6.29 do Linux, o subsistema *cpuacct* é compilado no *kernel* habilitando-se a variável `CONFIG_CGROUP_CPUACCT` e é atribuído a uma hierarquia através da opção `cpuacct` da chamada de sistema `mount()`.

Este subsistema permite a contabilização do tempo de CPU utilizada pelos processos dos seus respectivos grupos, onde o tempo de CPU utilizado por processos de grupos filho também é contabilizado pelo grupo pai.

O subsistema *cpuacct* cria os seguintes arquivos de controle:

- `cpu_usage`: tempo de CPU acumulado, em nanosegundos, de todos os processos membros do grupo e de grupos filhos, desde a sua inclusão em cada grupo
- `cpu_usage_cpu`: divide o tempo informado no arquivo anterior por cada CPU do sistema
- `cpu_stat`: estatísticas de uso do tempo de CPU do grupo. Este arquivo apresenta a separação do tempo de CPU em espaço de usuário e em espaço de núcleo. A unidade utilizada é `USER_HZ`

Device Controller

Introduzido a partir da versão 2.6.29 do Linux, o subsistema *Device Controller* é compilado no *kernel* habilitando-se a variável `CONFIG_CGROUP_DEVICE` e é atribuído a uma hierarquia através da opção `devices` da chamada de sistema `mount()`.

Este subsistema monitora as chamadas de sistema `open` e `mknod` em arquivos especiais de dispositivos, podendo aplicar restrições de acessos a estes arquivos, baseando-se em listas de permissões de acesso (*blacklist* e *whitelist*) definidas pelo administrador da hierarquia.

O subsistema *devices* cria os seguintes arquivos de controle:

- `devices.allow`: arquivo de entrada dos atributos de controle de acesso para dispositivos permitidos
- `devices.deny`: arquivo de entrada dos atributos de controle de acesso para dispositivos não permitidos
- `devices.list`: listagem de todos os dispositivos com acesso permitido ao grupo

A sintaxe para escrita nos arquivos `devices.allow` e `devices.deny` é do tipo `<type> <major>:<minor> <access>` . Para o campo `<type>` são aceitos os valores `b`, `c` e `a` para, respectivamente, dispositivos de bloco, caractere e ambos. O segundo e terceiro campo recebem os números que identificam o dispositivo. Nestes campos, o caractere `*` é aceito e corresponde a todos os números daquele campo. Finalmente, o campo `<access>` recebe as regras de acesso da diretiva, podendo ser uma combinação dos valores `r` (leitura), `w` (escrita) e `m` (`mknod`).

O grupo raiz de uma hierarquia que utilize o subsistema `devices` inicia com acesso total (`rwm`) a todos os dispositivos do sistema. Um grupo filho herda as permissões do grupo pai e, não pode receber permissões de acesso a dispositivos aos quais o pai não possua acesso. Contudo, caso uma permissão comum a grupos pai e filho seja removida do grupo pai, ela não é automaticamente removida do grupo filho.

Freezer

Introduzido a partir da versão 2.6.28, o subsistema *Freezer* é compilado no *kernel* habilitando-se a variável `CONFIG_CGROUP_FREEZER` e é atribuído a uma hierarquia através da opção `freezer` da chamada de sistema `mount()`.

Este subsistema usa a infraestrutura do TCG para definir grupos nos quais é possível parar e reiniciar processos, sendo especialmente útil no gerenciamento de processos em lote, permitindo ao administrador agendar a execução de tarefas de acordo com a sua necessidade.

Outra aplicação é a criação de *checkpoints*[13] de grupos de aplicações, o que permite que sejam obtidas imagens consistentes dos processos de um grupo. Estas imagens podem ser utilizadas para retornar os processos a um estado consistente após uma falha ou mesmo para mover os processos entre diferentes nós de um *cluster*[14], por exemplo. Para gerar tais imagens, o grupo deve ser congelado, enquanto um outro processo usa as informações disponíveis em `/proc` ou através de outras interfaces de *kernel* para analisar o processo.

A utilização deste sistema não é equivalente a utilização de sinais do sistema operacional. As sequências de sinais `SIGSTOP` e `SIGCONT` nem sempre são suficientes para parar e reiniciar processos no espaço de usuário. Isso acontece porque, embora o sinal `SIGSTOP` não possa ser ignorado, ele pode ser observado pelo processo que o recebe (ou seja o processo sabe que recebeu tal sinal). Já o sinal `SIGCONT` pode ser tratado pelo processo que o recebe, fazendo com que ele haja de maneira diferente do esperado ao receber tal sinal. Portanto, um processo ao voltar a ser executado através do sinal `SIGCONT`, pode escolher proceder de maneira diferente do que normalmente o faria, uma vez que sabe que foi anteriormente parado através do sinal `SIGSTOP`. Um exemplo deste comportamento pode ser observado na aplicação `bash`. Um processo `bash` ao receber a sequência de sinais `SIGSTOP` e `SIGCONT` é finalizado, ao invés de simplesmente ser paralisado e voltar a executar normalmente. Esta é uma decisão de programação da aplicação, que é possível graças às características da manipulação de processos através de sinais ora apresentada. Desta forma, fica claro que a utilização de sinais não é eficaz para a manutenção de *checkpoints*.

[13] Técnica que permite que o estado de uma aplicação em um determinado momento seja salvo, permitindo avaliação e retomada da execução a partir daquele ponto

[14] Conjunto de computadores utilizados simultaneamente, de maneira coordenada, visando o atendimento de um único objetivo

O subsistema *freezer* cria um arquivo de controle, `freezer.state`, que ao ser lido informa o estado do grupo. A escrita neste mesmo arquivo é utilizada para modificar o estado do grupo. Este arquivo pode assumir os seguintes valores:

- `FROZEN`: processos do grupo estão congelados (não estão rodando)
- `THAWED`: processos do grupo estão sendo executados normalmente (não estão congelados)
- `FREEZING`: alguns processos do grupo estão congelados e outros rodando. Nos casos em que o subsistema não conseguir congelar todos os processos do grupo, a operação de escrita em `freezer.state` falha, retornando o valor `EBUSY` e o estado `FREEZING` passar a ser lido neste arquivo. Nestes casos, o usuário do subsistema pode escrever `THAWED` no arquivo de controle, para fazer com que todos os processos voltem a executar normalmente, ou escrever `FROZEN`, numa nova tentativa de congelar todos os processos do grupo.

Memory Resource Controller

Introduzido a partir da versão 2.6.29 do Linux, o subsistema *Memory Resource Controller* é compilado no *kernel* habilitando-se as variáveis `CONFIG_MEMCG` e `CONFIG_MEMCG_SWAP` e é atribuído a uma hierarquia através da opção `memory` da chamada de sistema `mount()`.

Este subsistema permite o controle do uso de páginas de memória (física e *swap*) por parte de grupos de processos. Desta forma, é possível estabelecer um mecanismo de isolamento onde aplicações que consomem muita memória possam ser limitadas a uma quantidade menor da memória total disponível.

O subsistema *Memory Resource Controller* cria os seguintes arquivos de controle:

- `memory.failcnt`: contador do número de tentativas de alocação de memória que falharam
- `memory.usage_in_bytes`: contator da quantidade de memória física consumida pelo grupo
- `memory.limit_in_bytes`: valor limite de uso de memória física pelo grupo
- `memory.soft_limit_in_bytes`: valor limite de uso de memória física pelo grupo. Este valor pode ser ultrapassado pelo grupo, desde que permaneça abaixo do limite estabelecido na variável anterior. Contudo, caso o sistema operacional precise de memória, ele irá solicitar dos grupos que tiverem extrapolado este limite

- `memory.max_usage_in_bytes`: consumo máximo de memória física registrado
- `memory.memsw.failcnt`: contador semelhante a `memory.failcnt`, aplicado a memória do tipo *swap*
- `memory.memsw.usage_in_bytes`: contador semelhante a `memory.usage_in_bytes`, aplicado a memória do tipo *swap*
- `memory.memsw.limit_in_bytes`: semelhante a `memory.limit_in_bytes`, aplicado a memória do tipo *swap*
- `memory.memsw.max_usage_in_bytes`: semelhante a `memory.max_usage_in_bytes`, aplicado a memoria do tipo *swap*
- `memory.force_empty`: variável de controle utilizada para zerar os contadores de uso de memória do grupo. Deve ser utilizada somente quando não existem processos atribuídos ao grupo (arquivo `tasks` deve estar vazio)
- `memory.swappiness`: variável de controle utilizada para habilitar ou desabilitar a capacidade do grupo de utilizar memória do tipo *swap*
- `memory.stat`: diversas estatísticas sobre o uso de memória
- `memory.use_hierarchy`: habilita contabilização hierárquica

HugeTLB Resource Controller

Introduzido a partir da versão 3.5 do Linux, o subsistema *HugeTLB Resource Controller* é compilado no *kernel* habilitando-se a variável `CONFIG_CGROUP_HUGETLB` e é atribuído a uma hierarquia através da opção `hugetlb` da chamada de sistema `mount()`.

Este subsistema implementa os mesmos mecanismos de controle sobre páginas de memória do *Memory Resource Controller* em páginas de memória do tipo *Huge TLB* [45] [43]. A sua utilização é semelhante àquele subsistema, através de arquivos de controle prefixados por `hugetlb`.

Network Priority Cgroup

Introduzido a partir da versão 3.3 do Linux, o subsistema *Network Priority Cgroup* é compilado no *kernel* habilitando-se a variável `CONFIG_NETPRIO_CGROUP` e é atribuído a uma hierarquia através da opção `net_prio` da chamada de sistema `mount()`.

Este subsistema permite um mapeamento de prioridades de tráfego gerado por diferentes aplicações, por *interface* de rede. A sua utilização é feita através dos seguintes arquivos de controle:

- `net_prio.prioidx`: valor inteiro utilizado no *kernel* para identificar o grupo. Este valor é definido automaticamente pelo subsistema, portanto o arquivo possui caráter informativo e é do tipo somente-leitura

- `net_prio.ifpriomap`: mapa com lista de prioridades estabelecidas para tráfego de rede originado pelos processos deste grupo. Para definir uma regra neste mapa, uma *string* do tipo `<interface> <prio>`, onde o primeiro campo representa o nome da *interface* de rede para a qual o prioridade está sendo definida e o segundo é um número inteiro da escala de prioridades, deve ser escrita neste arquivo de controle. Um exemplo válido seria: `eth0 5`.

Resource Counters

Introduzido a partir da versão 2.6.29 do Linux, o subsistema *resource counter* é compilado no *kernel* habilitando-se a variável `CONFIG_RESOURCE_COUNTERS`.

Diferentemente de outros subsistemas, esse não é um subsistema montável e pronto para uso, mas sim uma API que pode ser utilizada para o desenvolvimento de diversos subsistemas. Este subsistema tem como objetivo facilitar o gerenciamento de recursos de diferentes controladores, fornecendo atributos comuns de contabilização. Estes atribuídos são disponibilizados através de uma estrutura chamada `res_counter` e diversas funções para a manipulação desta.

A estrutura `res_counter` possui a seguinte forma:

```
struct res_counter {
     unsigned long long usage;
     unsigned long long max_usage;
     unsigned long long limit;
     unsigned long long failcnt;
     spinlock_t lock;
     struct res_counter *parent;
};
```

Onde os seus atributos representam as seguintes informações:

- `usage`: nível de consumo atual do recurso pelo grupo. A unidade deve ser definida pelo desenvolvedor do subsistema

- `max_usage`: valor máximo de consumo do recurso pelo grupo em uma curva histórica

- `limit`: valor limite para utilização do recurso. Um requisição de alocação de recursos capaz de exceder este limite deve ser negada pelo subsistema, garantindo sempre que `usage` seja menor ou igual a `limit`

- `soft_limit`: valor limite inicial para utilização do recurso. Um requisição de alocação de recursos capaz de exceder este limite deve ser permitida somente nos casos em que o limite máximo não seja excedido, ou seja, quando `usage` permanecer menor ou igual a `limit`
- `failcnt`: contador do número de tentativas de alocação do recurso que falharam
- `lock`: variável utilizada para garantir exclusão mútua no acesso à estrutura
- `*parent`: ponteiro para a estrutura `res_counter` do grupo pai. Utilizada para contabilidade hierárquica.

Desta forma, um subsistema para fazer uso desta API deve:

- incluir o cabeçalho da API (`linux/res_counter.h`)
- incluir uma variável do tipo `struct res_counter` na sua estrutura `task_group`
- incluir ganchos nos pontos onde são feitas alocação e liberação dos recursos a serem contabilizados/controlados pelo subsistema, de tal forma que, sempre que estes eventos ocorrerem os atributos de contabilização sejam atualizadas. A manipulação destes atributos deve ser feita através das funções definidas pelo `resource counter`, e não por meio de acesso direto às variáveis da estrutura
- incluir arquivos de controle do TCG para leitura dos parâmetros contabilizados e escrita dos limites a serem impostos

4.6 Conclusão

Este capítulo apresentou brevemente o funcionamento do sistema de controle de acesso do Linux, considerando sua arquitetura e funcionamento padrão (discricionário), além das possibilidades de extensão destas funcionalidades através aplicação de sistemas de controle de acesso do tipo mandatório. Foi apresentado o *framework* LSM, utilizado no Linux como ferramenta padrão para a implementação de diferentes modelos de controle de acesso.

Este capítulo abordou ainda outros recursos relacionados à segurança no ambiente do sistema operacional Linux: *SecurityFS*, um sistema de arquivos virtual utilizado como *interface* para módulos LSM (Seção 4.3); *Netfilter*, um *framework* para interceptação e filtro de pacotes de rede (Seção 4.4); e *Task Control Group*, um *framework* para agrupamento de processos e contabilização de uso de recursos por estes grupos, utilizado para monitorar o uso de recursos do sistema operacional e prevenir abusos (Seção 4.5).

Finalizada a revisão bibliográfica, a partir do próximo capítulo será apresentado a ferramenta desenvolvida neste trabalho.

Capítulo 5

Framework IMN

Nos Capítulos 2, 3 e 4 foram apresentados conceitos e definições pertinentes levantados na etapa de revisão bibliográfica deste trabalho. O Capítulo 2 apresentou a estrutura e funcionamento básico do sistema imunológico humano, fonte de inspiração de pesquisa para diversos trabalhos de pesquisa em segurança computacional, alguns dos quais foram brevemente descritos naquele capítulo. Dentre os trabalhos apresentados, a pesquisa realizada pelo projeto Imuno, do Laboratório de Segurança e Criptografia da Universidade Estadual de Campinas (LASCA - UNICAMP), motivou esta dissertação de mestrado.

Na Seção 2.2.4, foi apresentado o trabalho de De Paula [22], que propunha o desenvolvimento de um Sistema de Detecção de Intrusão (IDS - *Intrusion Detection System*) baseado no sistema imunológico humano e implementado no núcleo do sistema operacional GNU/Linux. Este sistema operacional, pelas características apresentados na Seção 3.1, foi o escolhido para as implementações dos trabalhos de pesquisa desenvolvidos pelo projeto Imuno. Por este motivo, os Capítulos 3 e 4 apresentaram a estrutura, funcionamento, recursos de segurança e ferramentas de gerenciamento do *kernel* Linux.

Neste capítulo, será apresentada a motivação, levantamento de requisitos e implementação do *framework* IMN. Na Seção 5.1, será apresentado o contexto e as dificuldades do projeto Imuno que motivaram a elaboração deste trabalho. Na Seção 5.2 será feita uma análise dos requisitos comuns aos trabalhos desenvolvidos no projeto, a partir de análise dos trabalhos apresentados na Seção 2.2. Na Seção 5.3 será apresentada a arquitetura e detalhes de implementação do *framework* IMN, trabalho fruto desta dissertação.

5.1 Motivação

Com o amadurecimento da pesquisa e o desenvolvimento de alguns protótipos de escopo reduzido, o projeto Imuno notou a necessidade de uma plataforma de desenvolvimento comum para os protótipos que vinham sendo trabalhados. Esta necessidade partiu

da constatação de que o processo de desenvolvimento dos protótipos era muito complexo, demandando grande esforço de desenvolvimento e manutenção de código. Essa realidade configurou uma das grandes barreiras no desenvolvimento do sistema imunológico geral proposto inicialmente, uma vez que muito esforço era dirigido para a implementação de pequenos requisitos, diminuindo assim o tempo de trabalho útil no desenvolvimento e implementação dos conceitos imunológicos.

Outra importante constatação, foi a grande dificuldade envolvida no processo de manutenção e atualização dos protótipos desenvolvidos para versões mais recentes do ambiente de desenvolvimento. Conforme visto anteriormente, estes protótipos eram em grande parte implementados em nível de *kernel* do Linux. A cada nova versão do *kenrel*, eram detectadas mudanças profundas na estrutura de alguns de seus componentes, o que, consequentemente, implicava em mudanças nas APIs utilizadas pelos protótipos do Imuno.

O primeiro impacto desta característica do ambiente de desenvolvimento, era um aumento da dificuldade relatada no parágrafo anterior: além da complexidade inerente ao desenvolvimento de ferramentas em nível de *kernel*, os pesquisadores ainda enfrentavam um grande esforço de programação a cada nova versão do Linux. Desta forma, o tempo demandado no desenvolvimento era somado ao tempo consumido em manutenção das ferramentas desenvolvidas, diminuindo ainda mais o tempo de trabalho útil na pesquisa dos conceitos imunológicos.

O segundo impacto, e talvez o mais considerável para a evolução da pesquisa, era o alto índice de abandono de código decorrente das constantes mudanças do Linux. Com a entrada de novos pesquisadores no projeto, e saída de outros, houve interrupção no processo de desenvolvimento dos protótipos. Quando da entrada de novos pesquisadores, após algum esforço de manutenção e atualização dos códigos existentes, notou-se uma tendência natural de abandono da base de código existente, e reimplementação dos conceitos pesquisados e introduzidos anteriormente.

Diante desta realidade, Martim Carbone [29] propôs o desenvolvimento de um *framework* a ser implementado em nível de *kernel*, sobre o qual os demais pesquisadores do projeto poderiam desenvolver, em espaço de usuário, os módulos do sistema imunológico por eles pesquisados. Desta forma, o processo de desenvolvimento seria simplificado. Além disso, as questões de manutenção e atualização das implementações de *kernel* inerentes às atualizações do Linux ficariam concentradas neste ponto da arquitetura Imuno, o *framework* Imunológico, podendo ser tratado de maneira isolada e por equipes independentes.

Outra importante vantagem da solução proposta por Carbone seria o aumento da capacidade de interoperação entre os módulos desenvolvidos pelos diferentes pesquisadores, uma vez que todos utilizariam essa mesma plataforma. O *framework* concebido por Carbone chegou a ser implementado na forma de um protótipo inicial de escopo reduzido,

com algumas das funcionalidades inicialmente propostas.

Este trabalho, portanto, teve como objetivo inicial a conclusão do protótipo desenvolvido por Carbone, o *framework* Imuno.

O resultado esperado é um *framework* funcional, estável, e em produção no ambiente do projeto Imuno. A visão do trabalho é implementar a solução de maneira tal que venha ao encontro das práticas de projeto adotadas pelos desenvolvedores do Linux, de forma que possa ser eventualmente aceito na versão oficial do *kernel*.

5.2 Requisitos

5.2.1 Metodologia

Para atender aos objetivos deste trabalho, em sua proposta inicial [15] foram definidas as seguintes etapas de atuação:

1. Atualização do protótipo de escopo reduzido desenvolvido por Carbone: resgatar o *framework* Imuno, atualizando-o para a versão corrente do Linux e restaurando o seu funcionamento;

2. Implementação de requisitos levantados mas não implementados: o protótipo de Carbone referenciou funcionalidades importantes para o *framework* que não haviam sido implementadas, ou estavam apenas parcialmente disponíveis. Estas funcionalidades, descritas na seção de sugestões de trabalhos futuros, deveriam então ser as primeiras novas funcionalidades a serem incorporadas ao *framework* Imuno atualizado;

3. Levantamento de novos requisitos para o *framework*: uma vez funcional e com os requisitos pré-estabelecidos implementados, a próxima etapa seria uma série de entrevistas com pesquisadores do projeto, de forma a levantar novos requisitos de funcionalidades que deveriam compor o rol de funcionalidades do *framework*;

4. Implementação das novas funcionalidades levantadas na etapa anterior.

5. Testes: exercícios de prova de funcionalidades e desempenho do *framework* desenvolvido.

Na Seção 5.2.2 serão descritas as atividades realizadas em cada uma destas etapas, bem como os requisitos levantados. A implementação destes requisitos e detalhamento da arquitetura e funcionalidades do *framework* serão descritas na Seção 5.3.

5.2.2 Levantamento de Requisitos

Etapa 1

Conforme apresentado na Seção 2.2.5 e ilustrado na Figura 2.10, o *framework* Imuno é um módulo LSM que implementa uma abstração capaz de permitir que diferentes LKMs registrem funções tratadoras para ganchos LSM. Esta funcionalidade foi batizada de Ganchos Multifuncionais e é manipulada através de uma nova chamada de sistema introduzida com o *framework* (chamada `imuno()`). A estrutura de ganchos multifuncionais é o cerne do Imuno.

Para garantir que apenas processos componentes do sistema de segurança imunológico possam utilizar as funcionalidades da arquitetura de ganchos multifuncionais, o *framework* Imuno implementa uma lógica de registro de processos através da chamada `imuno()` e da *interface* administrativa (sistema de arquivos virtual do tipo RelayFS). Nesta lógica, somente os processos registrados junto ao *framework*, ou "Processos Imunológicos", podem ter acesso aos recursos e dados da ferramenta.

Desta forma, além dos ganchos e lógica de registro de processos, o Imuno implementa uma barreira de auto-proteção para garantir que processos não imunológicos não possam interferir na segurança do sistema. Esta barreira utiliza os POSIX *Capabilities* para impedir que processos não imunológicos carreguem ou descarreguem LKMs, e implementa alguns tratadores em ganchos LSM relacionados a operações sobre processos e *inodes* (na forma de LKMs, como um módulo imunológico), visando garantir o isolamento de processos imunológicos e repositórios de seguros do sistema de segurança.

Outros importantes recursos do *framework* são:

- Ganchos UndoFS: *middleware* desenvolvido por De Paula [22] para disponibilização das diretivas `undo()` e `redo()` em sistemas de arquivo Linux;
- Ganchos Netfilter: *framework* do *kernel* Linux que disponibiliza ganchos em pontos do *kernel* onde é possível a interceptação do fluxo de pacotes de rede (Seção 4.4);
- CKRM: *framework* Linux[1] que implementa controle de recursos do sistema operacional baseado em classes de processos [50].

O primeiro passo para atualização do Imuno, desenvolvido para o Linux 2.6.12, para a versão 2.6.31 (versão do Linux do momento do início do desenvolvimento deste trabalho), foi desabilitar todos os recursos auxiliares daquele *framework* e compilar o LKM, juntamente com a lógica de ganchos multifuncionais, sobre a versão mais recente do Linux e carregar o módulo LSM neste sistema operacional. A conclusão com sucesso desta etapa representaria uma atualização das funcionalidades básicas do Imuno para o novo *kernel*.

[1] Projeto de pesquisa descontinuado, não chegou a fazer parte de versões oficiais do Linux

Este simples exercício de manutenção da ferramenta, ilustrou bem o obstáculo representado pela manutenção das ferramentas desenvolvidas em *kernel* pelo projeto Imuno: a partir da versão 2.6.24 do Linux, o *framework* LSM deixou de permitir que módulos de segurança fossem registrados como LKMs. A alegação dos mantenedores do sistema operacional era de que esta característica poderia comprometer a segurança do sistema como um todo, uma vez que processos maliciosos poderiam em tese remover módulos LSM em execução e também incluir os seus próprios módulos para mudar o comportamento dos sistemas de controle de acesso do Linux, permitindo um sem número de diferentes ataques (negação de serviço, roubo de informação, dentre outros).

Desta forma, a arquitetura sobre a qual o *framework* fora concebido, módulo de *kernel*, vira por terra. Diversas tentativas de ajuste no código do *framework* foram feitas na busca por alternativas à esta limitação imposta pelo LSM. Todas sem sucesso. Uma possível solução seria então alterar o código do Linux de forma a permitir novamente módulos LSM na forma de módulos de *kernel*, permitindo então o restabelecimento do funcionamento do Imuno.

A revisão feita sobre os trabalhos implementados pelo projeto Imuno (Capítulo 2) e o estudo apresentado sobre o funcionamento do *kernel* Linux (Capítulos 3 e 4) levaram à constatação de que implementações que alteram profundamente diferentes pontos do *kernel* são difíceis de manter. Estas soluções também possuem grandes restrições sobre seu aceite junto aos desenvolvedores do Linux, uma vez que estão sujeitas a falhas de implementação que não se limitariam à funcionalidade em si, podendo prejudicar também os pontos onde houveram alteração, o que poderia comprometer a estabilidade e segurança do sistema como um todo.

Por este motivo, este trabalho tem como diretriz a implementação de soluções com mínimo impacto no código do Linux e consoantes com as diretrizes e boas práticas adotadas pelos mantenedores do Linux, mantendo a visão de qualidade, usabilidade e possibilidade de aceite da ferramenta como um LSM do Linux oficial. A solução de modificar o *kernel* para permitir LSMs em forma de LKMs, portanto não foi explorada por não atender às diretrizes deste trabalho. Tal solução implica em mudanças profundas no *kernel* e é oposta à visão de projeto dos mantenedores do Linux, que haviam decidido por bem tornar os módulos LSM completamente *built-in*.

A Etapa 1 teve então seu escopo alterado: uma vez que uma simples atualização do *framework* Imuno não seria possível, os objetivos do trabalho deveriam então ser atingidos por meio de uma re-arquitetura e reimplementação. Esta nova implementação foi batizada, ***framework* IMN**.

O maior desafio deste novo *framework* era, mantendo as atuais características do LSM, permitir o carregamento dinâmico de funções tratadoras por módulos imunológicos. Este requisito foi atendido com sucesso através de uma nova implementação da infra-

estrutura de ganchos multifuncionais para este novo LSM. A Seção 5.3 irá detalhar a solução implementada e a arquitetura geral do IMN.

Etapa 2

Após minucioso estudo do trabalho desenvolvido por Martim e do protótipo Imuno, as funcionalidades abaixo foram levantadas:

- Funcionalidades Implementadas
  - Ganchos Multifuncionais via LSM implementado em LKM
  - Ganchos Netfilter
  - Interface administrativa via sistema de arquivos virtual RelayFS
  - Auto-proteção e repositório seguro
  - Controle de recursos via CKRM. Apenas controle de tempo de CPU disponibilizado
  - Ganchos Netfilter para monitoramento de pacotes de rede
  - UndoFS. Apenas ganchos de monitoramento de chamadas disponibilizados
- Funcionalidades Não Implementadas
  - Dump de páginas de memória para disco
  - Monitoração de recursos
  - Posicionamento de gancho multifuncional para todos os ganchos LSM
  - UndoFS. Foi implementado apenas a interceptação de chamadas e registro de dados. A reconstrução de arquivos não foi implementada
  - Criação de novos ganchos no *kernel*. Ex.: dados das chamadas `write()` e `truncate()` não são interceptados, uma vez que não guardam relação com controle de acesso, contudo seriam interessantes ao sistema imuno
  - Sistema mais eficiente para monitoramento de chamadas de sistema
  - Otimização de desempenho
  - Substituir ganchos UNDOFS por ganchos multifuncionais
  - Mecanismo de auto-proteção mais flexíveis. Possivelmente utilizando SELinux ou outra estratégia menos restritiva que o capabilities.

Todas essas funcionalidades passaram a compor, então, o conjunto inicial de requisitos para *framework* IMN. Inicialmente, contudo, somente a infraestrutura de ganchos multifuncionais modulares sobre o LSM IMN *builtin* foi implementado, conforme descrito anteriormente. Uma levantamento mais completo de requisitos foi feito antes da definição final da arquitetura da solução e de sua efetiva implementação.

Etapa 3

Neste ponto do trabalho, além de um estudo detalhado dos protótipos já realizados pelo grupo, seria de fundamental importância entrevistas sistemáticas com os demais integrantes do projeto, afim de conhecer melhor suas necessidades e dificuldades. Este conhecimento todo seria então compilado em uma proposta de solução que buscasse oferecer a maior parte possível dos recursos necessários, da forma mais simples possível.

O projeto Imuno, contudo, no ano de 2010, passou por mudanças significativas em sua estrutura. O foco da pesquisa acabou sendo direcionado para outras áreas de segurança de sistemas e o objetivo de construção de um sistema de segurança imunológico geral foi temporariamente adiado.

Desta forma, este trabalho foi parcialmente prejudicado pois não mais haveriam solicitações de requisitos por parte dos pesquisadores empenhados na construção de um sistema de segurança imunológico, tampouco haveria um *feedback* deles sobre as implementações iniciais do *framework* IMN, de forma a contribuir com sugestões de melhorias.

Por outro lado, toda a pesquisa sobre o *kernel* Linux e suas questões relativas à controle de acesso, controle de recursos e segurança em geral, motivaram o autor deste trabalho a seguir com o seu desenvolvimento. Primeiramente, já havia ampla disponibilidade de requisitos necessários para uma solução imunológica. Além disso, a generalidade da solução que estava sendo concebida poderia ser utilizada como plataforma para o desenvolvimento de outras soluções de segurança, não necessariamente relacionadas com os princípios de funcionamento do sistema imunológico humano.

Assim sendo, este trabalho prosseguiu com o estudo detalhado dos protótipos desenvolvidos pelo projeto Imuno, buscando extrair suas principais funcionalidades e observando atentamente os pontos em que eles falharam, seja na implementação quanto na manutenção, de forma a evitar a repetição de erros e elevar a qualidade do *framework* em desenvolvimento.

O primeiro trabalho analisado, ***"Forense computacional e sua aplicação em segurança imunológica"*** [30], o qual propunha uma arquitetura extensível para analisadores forenses, desenvolvida por Marcelo Reis, tal como apresentado na Seção 2.2.2, realiza análise forense sobre informações coletadas em máquinas suspeitas de ataque. A proposta do trabalho era implementar um protótipo para análise automática. Este

protótipo foi dividido em dois módulos: coletor de informações e analisador forense automatizado.

Este protótipo, contudo, não chegou ter uma versão funcional. O trabalho concentrou-se na introdução dos conceitos imunológicos, conceitos de detecção de intrusão e forense computacional, bem como em uma detalhada especificação de uma arquitetura extensível para análise forense automatizada.

Uma característica interessante do esforço de programação do protótipo AFA, é que ele implementa totalmente o módulo de coleta de informações. Este módulo, apesar de fundamental, uma vez que reúne toda a informação necessária para a análise forense proposta, pode ser considerado como um ferramental para o exercício da pesquisa realizada.

Nota-se naquele trabalho, uma extensa busca por mecanismos capazes de proceder com toda a coleta de informações necessária para atendimento do analisador. A dissertação de mestrado fruto daquela pesquisa [30] possui inclusive um extenso apêndice das diferentes estratégias adotadas ou implementadas para obtenção de tais informações. Este grande trabalho pode ter sido um fator determinante para a impossibilidade de conclusão da implementação do analisador proposto.

Daquele trabalho portanto, foram extraídos os principais **requisitos para detecção de intrusão** desejados para o *framework* IMN, a saber:

- Informações de memória principal (*dump* de memória)
- Monitoramento de sinais enviados a processos
- Monitoramento de tráfego de rede
- Informações sobre processos
- Arquivos abertos por processos
- Informações de estados de interfaces e conexões de rede
- Informações de tabelas e cache de roteamento
- Cache de resolução de endereços MAC
- Cache de resolução de nomes
- Listagem de módulos de *kernel* carregados
- Espelhamento de discos para análise forense
- Acesso a informações de *inode*

Boa parte dos requisitos acima são atendidos por ferramentas de espaço de usuário, com permissões de *root*. O papel do IMN, portanto, deve ser o de proteger e garantir a integridade destas ferramentas.

O trabalho desenvolvido por Diego Fernandes, ***"Resposta Automática em um Sistema de Segurança Imunológico"*** [20], também possui uma série de requisitos que devem conferir ao sistema de segurança imunológico a capacidade de limitar a ação de um possível ataque.

O sistema proposto por Marcelo utiliza agentes móveis em uma arquitetura distribuída. Em última instância, o protótipo faz uso de *"componentes locais de reação"*, capazes de efetivamente atuar (responder) sobre os processos suspeitos. Naquele trabalho, há uma clara divisão entre a concepção do sistema de segurança imunológico, ou seja, utilização de agentes móveis para implantação de mecanismos imunológicos de resposta primária e secundária e o ferramental necessário para atuar de maneira desejada sobre os processos investigados, os componentes locais de reação.

Estes componentes locais são foco do *framework* IMN e devem ser capazes de atuar em nível de processo e em nível de rede, disponibilizando as funcionalidades abaixo listadas, as quais compõem o conjunto de **requisitos para mecanismos de resposta**:

- Nível de processos:
  - Limite de uso de memória: capacidade de limitar a quantidade de memória física utilizada por um processo
  - Limite de consumo de CPU: capacidade de limitar o tempo de CPU utilizado por um processo
  - Número máximo de conexões: capacidade de limitar o número de conexões abertas por um processo
  - Quantidade de processos filhos: capacidade de limitar o número de processos filhos criados por um processo
  - Limite transações de acesso a disco: capacidade de limitar a leitura ou escrita de dados em disco, em termos de volume de dados, por um processo
  - Limite de acesso a disco: capacidade de limitar operações de leitura ou escrita em disco, em termos de permissões, por um processo
  - Finalização de processos: capacidade de finalizar processos
  - Paralisação de processos: capacidade paralisação e eventual continuação na execução de processos
- Nível de rede:

  - Bloqueio de tráfego: capacidade de bloquear tráfego por máquina, serviço, protocolo ou porta
  - Limite de conexões: capacidade de limitar o número de conexões por serviço porta
  - Bloqueio de tráfego por usuário: capacidade de bloquear tráfego de rede por usuário

Na implementação do seu protótipo, Marcelo desenvolveu em nível de *kernel* Linux vários mecanismos de reação para atender aos requisitos da solução:

- Nível de rede: iptables + *patch* de *kernel* contabilização de conexões simultâneas
- Nível de processos:
  - `cpulimit`: limitação de uso de CPU (diretiva `cpulimit()` desenvolvida por Diego sobre o escalonador O(1))
  - `conectionlimit`: limitação do número de conexões (feito via iptables)
  - `fslimit`: monitoramento de *syscalls* de manipulação de sistemas de arquivos. Implementado via LKM que cria uma nova *syscall* para forçar políticas sobre as chamadas de manipulação de sistemas de arquivos
  - `kill` e `stop`: ações de emergência feitas via sinais do Linux

Os requisitos de controle de recursos introduzidos passam a compor o conjunto de requisitos do *framework* IMN.

Apesar de funcional a nível de pesquisa, este protótipo traz consigo algumas características nocivas à manutenção e evolução do seu código em um sistema de segurança imunológico geral, a saber:

- Aplica um *patch* de terceiros sobre um componente do *kernel* (contabilização de conexões Netfilter): ferramentas em desenvolvimento e que não fazem parte do *kernel* costumam sofrer grande alterações entre versões e estão sob constante risco de simplesmente serem abandonadas, uma vez que são mantidas por desenvolvedores isolados, e não pela comunidade Linux;
- Inclui uma nova chamada de sistema no Linux: conforme visto no Capítulo, 3, a introdução novas chamadas de sistema deve ser evitada;
- Altera o código do escalonador do Linux para atendimento de um requisito específico: trata-se de uma profunda alteração em um ponto de grande importância do sistema operacional. Além de difícil de manter, esta abordagem envolve risco de comprometimento da estabilidade do sistema como um todo.

Outro ponto importante sobre o protótipo de Marcelo, é que ele não implementa qualquer mecanismo de auto-proteção, uma vez que este seria um esforço de programação que desviaria do foco da pesquisa, e esta limitação fora considerada aceitável por tratar-se de um protótipo com objetivo de comprovar a eficácia da utilização de agentes móveis para implantação de um módulo de resposta automática de um sistema de segurança imunológico. Um sistema geral, contudo, deve possuir mecanismos de auto-proteção capazes de regular o funcionamento do sistema mesmo na ocorrência de ataques.

Por fim, o protótipo ADenoIDS, de De Paula [22], introduz vários requisitos funcionais para o *framework* IMN. Abaixo, estes requisitos serão apresentados no contexto de cada módulo do sistema proposto. Módulos não implementados ou puramente imunológicos, não serão apresentados abaixo, uma vez que não introduzem requisitos ao IMN.

- Fonte de informação (ADDS):
  - Monitoramento de processos: operações sobre sistemas de arquivo, criação e execução de processos (processos filhos devem ser monitorados), inserção e remoção módulos de *kernel*, IPC (sinais e sockets);
  - Monitoramento de tráfego de rede;
- Agente de resposta inata (ADIRA): bloqueia todos os processos em espaço de usuário (SIGSTOP), recupera o sistema (UNDOFS), mata processos invasores, desbloqueia o restante dos processos bloqueados (SIGCONT) e utiliza o UNDOFS para recuperar alterações feitas no sistema de arquivos;
- Repositório para suporte forense (ADFSR): repositório seguro para armazenamento de informações do ADDS;
- Ferramentas de apoio:
  - Auto-proteção. Bloqueio de sinais de processos normais para processos ADENOIDS e proteção dos arquivos de configuração e log do sistema;
  - UNDOFS: `middleware` do VFS que implementa operações `undo()` e `redo()`.

Assim como os demais protótipos desenvolvidos no projeto, o ADenoIDS obtém sucesso na implementação dos seus requisitos através de profundas alteração no *kernel* Linux. Posteriormente, o protótipo deixou de ser mantido devido à sua complexidade de implementação.

Um resumo das funcionalidades, pontos positivos e principais ofensores de implementação/manutenção dos trabalhos desenvolvidos pelo projeto Imuno é ilustrado na Figura 5.1. O conjunto de requisitos consolidado para o *framework* IMN é apresentado na Seção 5.2.3.

Figura 5.1: *Timeline* do Projeto Imuno.

Etapas 4 e 5

A implementação do *framework* IMN será apresentada na Seção 5.3. Os testes praticados com a ferramenta serão apresentados no Capítulo 6.

5.2.3 Requisitos Consolidados

Os requisitos levantados na Seção anterior foram consolidados em quatro grandes grupos: monitoramento, resposta, auto-proteção e administração. Abaixo os requisitos de cada grupo são detalhados:

1. **Monitoramento**:
   - Processos:
     - Operações sobre sistemas de arquivo;
     - Criação e execução de processos;
     - Inserção e remoção de módulos de kernel;
     - IPC (Sinais e Sockets);
   - Rede:
     - Serviços;
     - Protocolos ;

  - Número de Conexões;
  - Volume de dados;
- Host:
  - *dump* de memória;
  - informações sobre processos;

2. **Resposta:**
   - Bloqueio e desbloqueio de processos de espaço de usuário;
   - Finalização de processos invasores;
   - Recuperação de alterações feitas no sistema de arquivos durante um ataque;
   - Controle de recursos (bloqueios/limitações/garantias) em nível de processo;
3. **Auto-proteção:**
   - Repositório Seguro para suporte forense;
   - Proteção de caminhos específicos do sistema de arquivos;
   - Proteção de processos imunológicos;
   - Proteção do *framework* IMN e seus componentes externos (LKMs imunológicos, e outras ferramentas de *kernel*);
4. **Administração:**
   - Interface de comunicação eficiente entre o *framework* e os módulos imunológicos (espaço de *kernel* e espaço de usuário);
   - Mecanismo de registro e desregistro de módulos imunológicos;
   - Mecanismo de ativação e desativação de auto-proteção;
   - Mecanismo de registro de tratadores de ganchos LSM por módulos imunológicos (LKMs imunológicos).

5.3 Arquitetura

O *framework* IMN possui uma arquitetura geral composta basicamente de três diferentes tipos de componentes: o *framework*, LKMs imunológicos e módulos imunológicos. A Figura 5.2 ilustra tal arquitetura.

O *framework* é uma abstração em nível de *kernel* Linux, implementado como um módulo LSM, que disponibiliza uma série de recursos com o objetivo de ser base comum

para o desenvolvimento de módulos imunológicos. Os módulos imunológicos, por sua vez, são aplicações em espaço de usuário que devem implementar componentes do sistema de segurança imunológico, fazendo uso dos recursos disponibilizados pelo *framework* IMN e demais recursos do sistema operacional GNU/Linux. Já os LKMs imunológicos são módulos de *kernel* que definem funções tratadoras para ganchos LSM que podem ser registradas por um módulo imunológicos, fazendo uso da funcionalidade de Ganchos Multifuncionais disponibilizada pelo IMN.

O objetivo principal desta ferramenta, tal como descrito na Seção 5.1, é a simplificação do processo de desenvolvimento, manutenção e integração de diferentes protótipos de módulos de segurança imunológica desenvolvidos pelo Projeto Imuno, que por sua vez busca o desenvolvimento de um sistema de segurança imunológico geral. Este trabalho tem como objetivo, portanto, compor o cerne de tal solução. Este objetivo foi perseguido através da evolução do *framework* Imuno, desenvolvido por Carbone [29], e criteriosa análise de requisitos, resultados e dificuldades dos trabalhos desenvolvidos pelos pesquisadores do projeto.

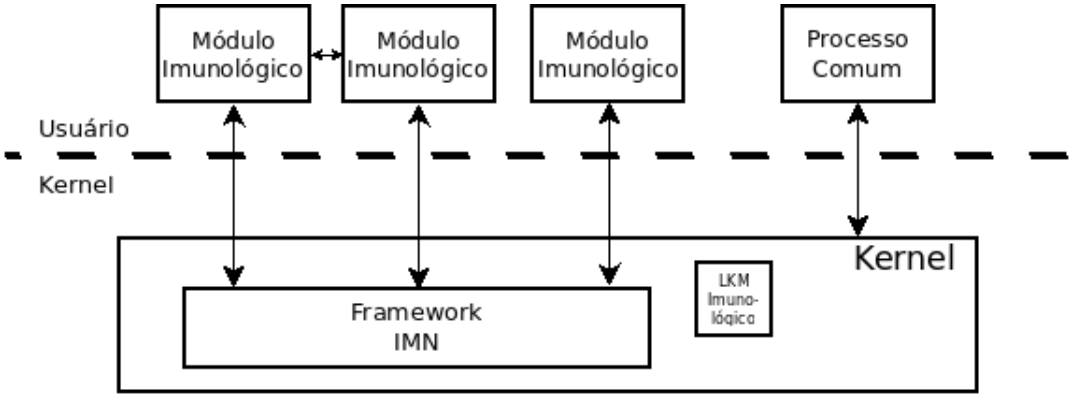


Figura 5.2: Arquitetura geral do *framework* IMN.

Como resultado de tal análise, além do levantamento dos requisitos realizado na Seção anterior, foi dada especial atenção à questão de manutenção do código a ser desenvolvido, principalmente na identificação e eliminação dos principais ofensores encontrados nas operações de atualização das ferramentas para novas versões do Linux.

Os protótipos desenvolvidos pelo projeto Imuno, ilustrados na Figura 5.1, possuem dois ofensores comuns: inclusão de novas chamadas de sistema ao Linux e utilização de ferramentas externas ao Linux e em fase de desenvolvimento (algumas delas desenvolvidas pelos próprios pesquisadores do Projeto Imuno). O primeiro ofensor, apesar de não representar um grande impacto de manutenção, representa uma barreira para a inclusão da ferramenta em versões oficiais do Linux.

O uso de ferramentas em desenvolvimento, por sua vez, foi considerado o maior ofensor de manutenção, pois estas ferramentas, por estarem em fase de desenvolvimento estão sujeitas a grandes alterações entre versões, abandono de projeto, ou mesmo rejeite de inclusão no Linux. Estas ferramentas, portanto, representam um grande obstáculo de manutenção dos protótipos e podem afastar ainda mais eles de um eventual aceite junto ao Linux. A Tabela 5.1 ilustra as ferramentas externas utilizadas em alguns protótipos do projeto e as dificuldades de manutenção decorrentes destas escolhas.

Protótipo	**Ferramenta Auxiliar**	**Ofensor**
Response Agent	Diretiva `cpulimit()` para O(1)	Ferramenta altamente invasiva
		Ferramenta deixou de ser mantida pelo autor
		A partir da versão 2.6.23 o Linux passou a utilizar o escalonador CFS, eliminando a utilidade deste *patch*
ADenoIDS	UNDOFS	Ferramenta deixou de ser mantida pelo autor
Framework Imuno	CKRM	Ferramenta deixou de ser mantida pelo autor
	RelayFS	Ferramenta deixou de ser mantida pelo autor
	UNDOFS	Carbone assumiu em seu trabalho a tarefa de atualização desta ferramenta. Apenas umas atualização parcial foi feita. A ferramenta foi então novamente abandonada

Tabela 5.1: Exemplos de ofensores de manutenção de protótipos do Projeto Imuno decorrentes de implementações invasivas ou utilização de ferramentas em desenvolvimento.

Em decorrência do aprendizado destas experiencias de desenvolvimento, o IMN tem como **regras de projeto** a implementação de soluções pouco invasivas e utilização de ferramentas Linux consolidadas, visando principalmente a clareza do código e facilidade de manutenção do mesmo. Desta forma, funcionalidades requeridas pelo projeto Imuno somente são atendidas pelo IMN nos casos em que for possível disponibilizá-las sem violar estas regras. Requisitos não atendidos por este motivo, serão devidamente apontados. Necessidades urgentes deverão ser implementadas e mantidas pelos pesquisadores até o momento que uma solução semelhante seja incluída no Linux, quando passará a compor o IMN.

Os recursos com implementação pouco invasiva desenvolvidos para o IMN são:

- Infra-estrutura de ganchos multifuncionais
- Mecanismos de auto-proteção

- Interface administrativa com troca de informações de espaço de *kernel* para espaço de usuário sem utilização de novas chamadas de sistema

Outros recursos externos são acoplados ao IMN através da garantia de auto-proteção sobre estes recursos, garantindo que apenas processos imunológicos possam fazer uso de tais ferramentas.

Dentro dessa visão da arquitetura do IMN, ilustrada na Figura 5.3, tem-se uma boa cobertura dos requisitos levantados através de uma implementação pouco invasiva e utilização de ferramental consolidado e estável no ambiente Linux. Eventuais requisitos indisponíveis e que não foram implementados pelo IMN por não atender às regras de projeto, ilustrados pelos blocos pontilhados na Figura 5.3, podem ser implementados e mantidos à parte por desenvolvedores de módulos imunológicos até que uma solução equivalente seja disponibilizada na versão oficial do Linux, quando passará a fazer parte do *framework* IMN.

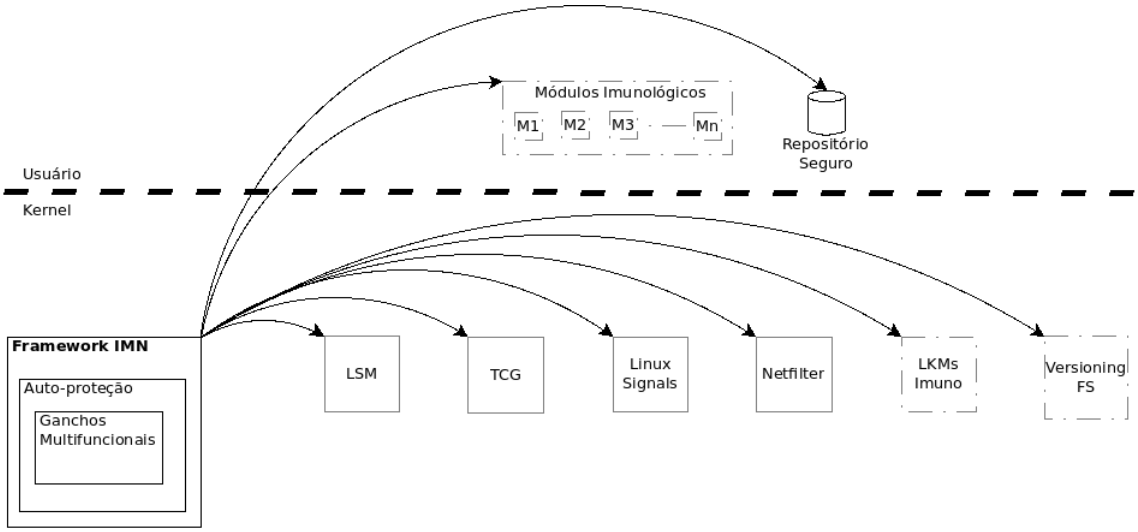


Figura 5.3: Arquitetura do *framework* IMN, outra visão.

A Figura 5.4 ilustra detalhadamento a arquitetura do IMN, bem como a relação entre os seus componentes. A implementação desta arquitetura é discutida na Seção 5.4.

5.4 Implementação

Nas seções a seguir, será detalhada a implementação dos diferentes componentes do *framework* IMN.

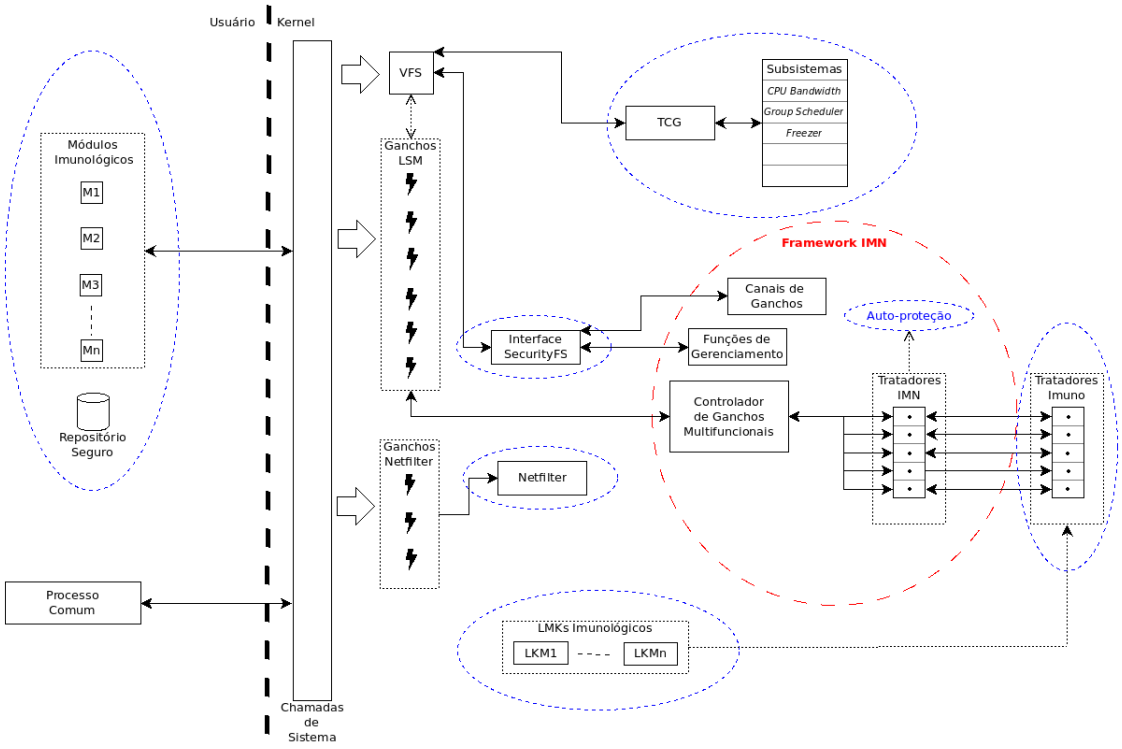


Figura 5.4: Arquitetura do *framework* IMN.

5.4.1 Ganchos Multifuncionais

Os ganchos do *framework* LSM, conforme visto na Seção 4.2, estão espalhados por todo o *kernel* Linux em pontos relacionados a operações de controle de acesso. Estes ganchos são de grande utilidade na implementação do sistema de segurança imunológico geral, uma vez que permitem fácil acesso a pontos de monitoramento e controle de permissões sobre de operações de processos em nível de *kernel*, tais como operações sobre arquivos, criação de processos, mecanismos de IPC, dentre outros.

O LSM, contudo, admite o registro de apenas um único módulo de segurança por instância de execução do Linux. Desta forma, um usuário do sistema operacional deve escolher entre um dos módulos LSM disponíveis ou mesmo desenvolver o seu. Não é possível, contudo, a utilização concomitante de diferentes módulos LSM. O administrador do sistema deve analisar dentre as opções disponíveis, e escolher aquela que melhor o atende. Por este motivo, uma implementação de um sistema de segurança imunológico admitiria que apenas um módulo imunológico fizesse uso dos ganchos LSM. Os demais deveriam ser implementados sem a utilização destes recursos.

A arquitetura do sistema de segurança imunológico geral proposta pelo projeto Imuno,

por outro lado, conforme apresentado na Seção 2.2.4, prevê uma implementação modular deste sistema, onde cada módulo representaria um componente do modelo biológico. Desta forma, a utilização de ganchos ficaria limitada a um único módulo imunológico. Este módulo, por sua vez, provavelmente utilizaria apenas um subconjunto dos ganchos para implementação do seu escopo. Os demais módulos do sistema deveriam então buscar outros mecanismos de implementação, que não fizessem uso dos ganchos LSM.

Visando contornar esta limitação e permitir a diferentes módulos do sistema de segurança imunológico o acesso a ganchos do LSM, Carbone concebeu a funcionalidade batizada de "Ganchos Multifuncionais" [29].

O *framework* IMN é implementado como módulo LSM, ou seja, registra-se junto ao *framework* LSM no momento da inicialização do sistema, passando a ter acesso total a todos os ganchos daquele *framework* e impedindo que outros módulos LSM sejam utilizados naquela instância de execução. O IMN não utiliza os ganchos LSM para implementar um modelo completo de controle de acesso mandatório, utilizando apenas alguns ganchos para implementação do seu mecanismo de auto-proteção (Seção 5.4.2). Portanto, o IMN implementa a arquitetura de ganchos multifuncionais proposta por Carbone para disponibilizar o acesso a ganchos LSM por parte de módulos imunológicos.

A Figura 5.5 ilustra o funcionamento da arquitetura de ganchos multifuncionais.

A estrutura dos ganchos multifuncionais é implementada pelo componente "Controlador de Gancho". Um módulo de segurança imunológica deve utilizar a *interface* administrativa do IMN (Seção 5.4.3) para registrar-se a um determinado gancho LSM. Uma vez registrado, um módulo imunológico pode definir diversas funções tratadoras para este gancho. Não é possível o registro de mais de um módulo imunológico por gancho, contudo é possível que um módulo registre-se em vários ganchos, desde que nenhum deles já esteja registrado para outro módulo. As funções tratadoras devem ser implementadas em LKMs (chamados de LKMs imunológicos) e ter os seus nomes exportados através da macro `EXPORT_SYMBOL()`.

É possível ainda a um módulo registrar-se em ganchos que já são utilizados pelo IMN. Nestes casos, a função tratadora interna do *framework* é executada primeiro e, caso o acesso não seja negado por ela, as funções tratadoras registradas pelo módulo imunológico do gancho são invocadas.

O IMN define funções tratadoras simplificadas para todos os ganchos do LSM[2]. Quando um determinado gancho LSM é alcançado no *kernel*, o controle de execução é desviado para para o tratador simplificado correspondente àquele gancho. O tratador, por sua vez, aciona o Controlador de Ganchos do IMN. Os tratadores simplificados do IMN possuem

[2]Todos os ganchos LSM disponibilizados pelo IMN com suas respectivas assinaturas de função e uma breve descrição de funcionamento estão listados no Apêndice A

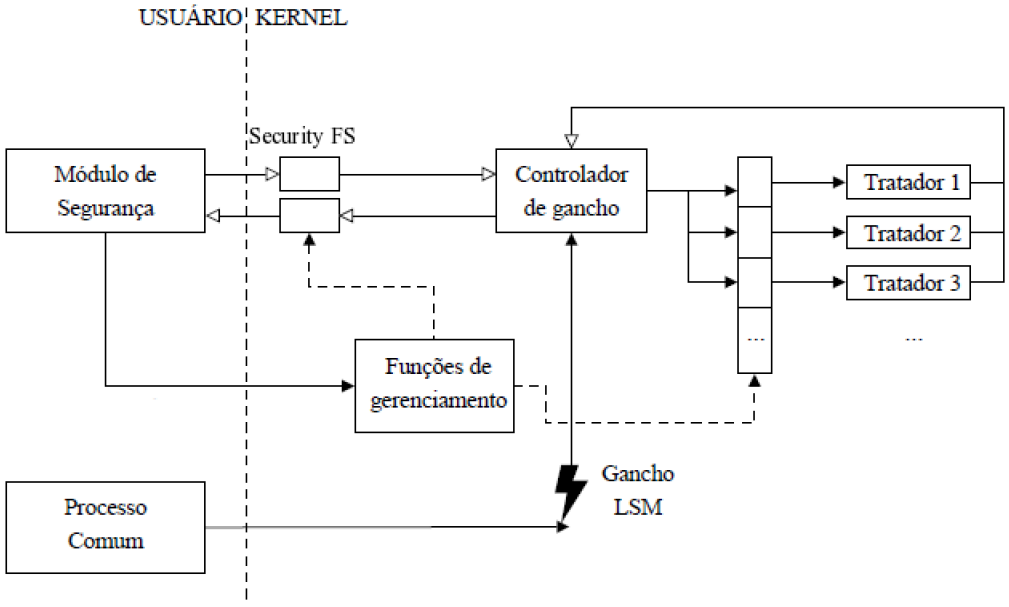


Figura 5.5: Estrutura de um gancho multifuncional.

a seguinte forma[3]:

```
int imn_file_open(struct file *file, const struct cred *cred)
{
   return imn_hook_start(ID_FILE_OPEN, file, cred);
}
```

Desta forma, sempre que um gancho LSM é alcançado, o fluxo de execução é desviado para o controlador de ganchos, o qual recebe uma lista variável de parâmetros, de acordo com o gancho alcançado. O controlador de ganchos, então, verifica se existem funções tratadoras registradas para aquele determinado gancho e desvia o fluxo de execução para o primeiro tratador, repassando os seus parâmetros e ponteiros para os canais de entrada e saída do gancho.

Após o retorno do primeiro tratador, o controlador de ganchos itera sobre os demais tratadores, de acordo com o modo de execução definido no momento do registro do gancho, pelo módulo imunológico, até que o último tratador seja atingido ou até que um

[3]A função `imn_hook_start()` implementa o controlador de ganchos do IMN

dos tratadores retorne um valor que interrompa a sequência de execução, permitindo ou negando o acesso ao recurso.

O valor retornado por cada tratador de um determinado gancho controla o comportamento do controlador de ganchos multifuncionais. Quando um valor menor ou igual ao valor -2 é retornado por um tratador, o controlador encerra o processo de iteração sobre os tratadores e retorna este número negativo para o LSM, o que implica em negação de acesso ao objeto no contexto em questão. O valor de retorno -1 indica ao controlador que a iteração sobre tratadores deve ser interrompida e o valor 0 deve ser retornado ao LSM, permitindo o acesso ao objeto. Valores de retorno maiores ou igual a 0 indicam ao controlador de ganchos qual tratador deve ser acionado em seguida. Esta última opção é válida apenas para os modos de execução 1 e 2.

O modos de execução dos ganchos multifuncionais são os seguintes:

- 0: executa os tratadores registrados para um determinado gancho do primeiro ao último, sequencialmente, até que o último tratador seja alcançado ou um dos tratadores retornem um valor negativo;

- 1: executa o primeiro tratador registrado. O valor retornado, caso maior ou igual a zero, determina qual o próximo tratador a ser executado. A iteração entre os tratadores só termina quando um dos tratadores retornar um valor negativo;

- 2: executa o primeiro tratador registrado e, após o retorno deste tratador, paralisa a execução do processo que invocou o gancho, através da função `set_current_state()` com o parâmetro `TASK_UNINTERRUPTIBLE`. Durante o período em que o processo não está sendo executado, o módulo imunológico que registrou o gancho pode ler os dados da função tratadora escrita no canal de saída do gancho na *interface* do IMN e decidir qual o próximo tratador a ser executado, escrevendo no canal de entrada do gancho o inteiro identificador do tratador a ser acionado. O controlador de ganchos então coloca o processo invocador novamente na fila de processos executáveis (`wake_up_process()`) e em seguida invoca o tratador cujo índice foi escrito pelo módulo imunológico no canal de entrada do gancho. O processo continua até que um tratador retorne valor negativo.

A utilização dos modos de execução 1 e 2 requer muito cuidado por parte dos desenvolvedores do sistema de segurança imunológico pois está sujeita à ocorrência de *loops* infinitos entre os tratadores registrados, uma vez que o controlador de ganchos só interrompe a iteração no nomento em que um dos tratadores retornar uma valor negativo.

Uma importante consideração a ser feita sobre a implantação da arquitetura de Ganchos Multifuncionais no *framework* IMN, e que difere sensivelmente da implementação do *framework* Imuno, é que ela não envolve a inclusão de uma nova chamada de sistema ao

Linux, tampouco utiliza ferramentas estranhas ao *kernel* (*patches* de terceiros) em sua implementação. O registro de tratadores para ganchos é feito diretamente sobre arquivos de controle da *interface* do IMN. Quando um módulo imunológico registra-se em um gancho, um arquivo de controle é criado para aquele gancho. Este arquivo, chamado de "canal de entrada e saída", é o ponto de troca de informação entre o IMN e os módulos imunológicos (espaços de núcleo e usuário).

Em espaço de núcleo, o IMN e os LKMs imunológicos têm acesso a *buffers* que representam estes caracteres. Em espaço de usuário, estes canais são manipulados pelos módulos imunológicos como arquivos comuns.

Definição de funções tratadoras

A Função `target_for_mkdir()`, ilustrada abaixo, é uma possível implementação de um tratador a ser registrado para o gancho multifuncional `ID_MKDIR` do IMN:

```
int target_for_mkdir(va_list args, char *ochan)
{
        struct inode *dir;
        struct dentry *dentry;
        int mode;

        dir = va_arg(args, struct inode *);
        dentry = va_arg(args, struct dentry *);
        mode = va_arg(args, int);

        sprintf(ochan, "PID %d criou o diretório %s.\n",
             current->pid, dentry->d_name.name");
        return -1;
}
```

Esta função, uma vez definida em um LKM e registrada por um módulo imunológico junto ao *framework* IMN, irá monitorar as ocorrências da chamada de sistema `mkdir()` e registá-las no canal de saída do gancho. Um módulo imunológico de monitoramento poderia utilizar este tratador para gerar um arquivo de *log* das ocorrências desta chamada.

5.4.2 Auto-proteção

O mecanismo de auto-proteção do *framework* IMN tem como objetivo primário a proteção do sistema operacional de tal forma que este mantenha-se funcional mesmo na ocorrência de ataques bem sucedidos.

Como o sistema de segurança imunológico deve ter a capacidade de detectar e responder a tais ataques, é fundamental que o mecanismo de auto-proteção do IMN seja capaz de garantir total proteção aos processos imunológicos, aos repositórios de informação forense (repositórios seguros), aos componentes externos do *framework* e ao próprio IMN, pois, caso algum destes elementos seja comprometido, o ataque pode conseguir inibir uma atuação eficaz do sistema de segurança.

Desta forma, o mecanismo de auto-proteção do IMN utiliza a arquitetura de ganchos multifuncionais apresentada na Seção 5.4.1 para proteger os diferentes componentes do sistema de segurança imunológico. Por questões de administração do IMN, este mecanismo pode ser habilitado e desabilitado através da *flag* global `selfprot`, facilmente manipulável através da *interface* administrativa do *framework*, tal como será apresentado na Seção 5.4.3.

A proteção dos componentes do sistema de segurança imunológico é feita utilizando-se diversos conjuntos de ganchos LSM. A seguir serão apresentados estes conjuntos e os diferentes componentes protegidos pelos seus tratadores.

Ganchos de operações sobre processos

De acordo com a arquitetura do *framework* IMN, apresentada na Figura 5.4, os módulos imunológicos são os únicos processos em espaço de usuário que são considerados componentes do sistema de segurança imunológico. Desta forma, a abordagem utilizada na proteção dos módulos imunológicos foi a criação de tratadores de auto-proteção para ganchos LSM de operações sobre processos. Apenas ganchos em funções capazes de alterar de alguma forma o comportamento ou estruturas de processos foram consideradas.

Estes tratadores negam o acesso em operações sobre processos sempre que o mecanismo de auto-proteção estiver ativada e um sujeito não-imunológico tentar atuar sobre um processo imunológico. Os ganchos LSM utilizados são os seguintes:

- `security_task_kill()`: intercepta solicitações feitas em espaço de usuário através da chamada de sistema `kill()`, permitindo a checagem de permissões antes do envio de sinais a um dado processo `p`;

- `security_task_setpgid()`: intercepta solicitações feitas em espaço de usuário através da chamada de sistema `setpgid()`, permitindo a checagem de permissões antes da alteração do GID de um dado processo `p`;

- `security_task_setnice()`: intercepta solicitações feitas em espaço de usuário através da chamada de sistema `nice()`, permitindo a checagem de permissões antes da alteração do nível de prioridade de escalonamento de um dado processo `p`;

- `security_task_setioprio()`: intercepta solicitações feitas de espaço de usuário através da chamada de sistema `ioprio_set()`, permitindo a checagem de permissões antes da alteração do nível de prioridade em operações de entrada e saída de um dado processo `p`;

- `security_task_setrlimit()`: intercepta solicitações feitas a partir do espaço de usuário através da chamada `setrlimit()`, permitindo a checagem de permissões antes da alteração de valores limites para diferentes recursos de um dado processo `p`, tais como valores limites para prioridade de escalonamento (`nice`), tamanho de pilha, fila de sinais pendentes, dentre outros;

- `security_task_setscheduler()`: intercepta solicitações feitas a partir do espaço de usuário através da chamada `schec_setscheduler()`, permitindo a checagem de permissões da alteração da política de escalonamento de um dado processo `p` (`SCHED_OTHER`, `SCHED_BATCH`, `SCHED_IDLE`, `SCHED_FIFO` e `SCHED_RR`).

Os tratadores de auto-proteção destes ganchos referenciam uma função de checagem de permissões do *framework* IMN, `imn_selfprot_check_pid()`, listada abaixo, que é a responsável pelo processo de tomada de decisão de acesso.

```
int imn_selfprot_check_pid(struct task_struct *p)
{
        spin_lock(&selfprot_lock);
        if (selfprot == 1) {
                spin_unlock(&selfprot_lock);
                if(imn_check_pid(p->pid) == 0)
                        if (imn_check_pid(current->pid) == 0)
                                        return 0;
                        else return -EACCES;
                else
                        return 0;
        } else {
                spin_unlock(&selfprot_lock);
                return 0;
        }
}
```

A função `imn_check_pid()` verifica se o processo correspondente ao PID informado está na lista de processos imunológicos do *framework*, retornando zero em caso afirmativo ou um inteiro negativo em caso contrário.

Ganchos de operações sobre `inodes`

Conforme visto na seção 3.5, o VFS é uma camada de abstração do Linux que funciona como *middleware* para implementação de diferentes sistemas de arquivos. O VFS define os elementos que compõem um sistema de arquivo, bem como toda a lógica de operações e manipulações sobre estes diferentes elementos. A implementação efetiva de cada uma das operações definidas pelo VFS para os diferentes objetos, por sua vez, fica a cargo do código dos diferentes sistemas de arquivo. Ações (chamadas de sistema) de sujeitos sobre objetos da hierarquia de arquivos do Linux, são interceptadas pelo VFS, que executa checagens e verificações e, em seguida, encaminha para efetiva execução do sistema de arquivos do objeto em questão.

Desta forma, para cada um dos ganchos LSM restritivos de operações sobre objetos VFS do tipo `inode`, o IMN registra um tratador que implementa parte do seu mecanismo de auto-proteção. Todos estes tratadores executam a mesma lógica de verificação: quando o mecanismo de auto-proteção estiver ativo, o acesso a objetos do tipo `inode` imunológico somente será permitido a processos imunológicos. A função `imn_selfprot_check_inode()`, abaixo, implementa esta lógica:

```
int imn_selfprot_check_inode(struct inode *inode)
{
        spin_lock(&selfprot_lock);
        if (selfprot == 1) {
                spin_unlock(&selfprot_lock);
                if(imn_check_inode(inode) == 0)
                        return 0;
                else {
                        if (imn_check_pid(current->pid) == 0 ||
                                imn_check_pid(current->parent->pid) == 0)
                                        return 0;
                        else return -EACCES;
                }
        } else {
                spin_unlock(&selfprot_lock);
                return 0;
        }
        return 0;
}
```

As funções `imn_check_inode()` e `imn_check_pid()` verificam, respectivamente, se o

`inode` e o processo informados são imunológicos, retornando zero em caso afirmativo ou um inteiro negativo em caso contrário.

A definição de `inodes` imunológicos é feita de maneira bem simples através do carregamento de uma listagem dos arquivos a serem protegidos, via *interface* do IMN. Esta abordagem permite a criação de **repositórios seguros**, atendendo de maneira direta este importante requisito do *framework*.

Ganchos do POSIX Capabilities

Usuários comuns do sistema operacional GNU/Linux têm poderes bastante restritos de atuação sobre os diversos componentes do sistema operacional. Por outro lado, o usuário privilegiado `root` possui plenos poderes sobre o sistema. Por este motivo, a implementação Linux do POSIX Capabilities, apresentada na Seção 4.1.1, busca limitar os poderes de processos com UID `root` a grupos de operações pré-determinados, chamados de *capabilities* (capacidades), numa tentativa de elevar a segurança do sistema como um todo.

O mecanismo de auto-proteção do IMN busca elevar ainda mais este nível de segurança, através da implementação de **controle de acesso mandatório sobre algumas destas capacidades**. Desta forma, processos não imunológicos não são capazes de executar determinadas ações, mesmo que tais processos possuam as devidas permissões DAC (máscaras de permissões, ACLs e capacidades). Ou seja, mesmo um processo `root`, com autorização para todas as capacidades do Linux em suas credenciais, não poderá atuar sobre os componentes desejados caso não seja um processo imunológico.

Esta implementação protege diferentes componentes do sistema Imuno. As capacidades restritas aos processos imunológicos e os respectivos objetivos de tais restrições são:

- CAP_SYS_MODULE: impedir que processos não imunológicos sejam capazes remover LKMs imunológicos ou carregar seus próprios LKMs, introduzindo código malicioso no *kernel*;

- CAP_SYS_RAWIO: impedir o acesso de processos não imunológicos a arquivos especiais de dispositivos que contenham informação de *kernel*. A limitação desta capacidade impede o acesso ao arquivo `/dev/kmem`, que contém a imagem da memória principal do computador;

- CAP_NET_ADMIN: impedir manipulação do *framework* Netfilter, componente externo do IMN utilizado para o atendimento de requisitos de monitoramento e resposta em nível de rede. A limitação desta capacidade impede ainda que processos não imunológicos executem outras atividades privilegiadas no subsistema de rede

do Linux, tal como alterações de configuração de interfaces de rede e tabelas de roteamento;

- CAP_SYS_PTRACE: impedir que processos não imunológicos observem ou alterem o comportamento de outros processos através do uso da chama de sistema `ptrace()`.

O controle de acesso mandatório sobre este subgrupo de capacidades Linux é obtido através da implementação do tratador `imn_selfprot_capable()` para o gancho LSM `security_capable()`. A implementação do tratador é bem simples e é ilustrada parcialmente abaixo:

```
int imn_selfprot_capable(va_list args, char *ochan)
{
...
        spin_lock(&selfprot_lock);
        if (selfprot == 1) {
                if (imn_check_pid(current->pid) == 0) {
                        spin_unlock(&selfprot_lock);
                        return 0;
                } else {
                        spin_unlock(&selfprot_lock);
                        if (cap == CAP_SYS_RAWIO)
                                return -EACCES;
                        else if (cap == CAP_NET_ADMIN)
                                return -EACCES;
                        else if (cap == CAP_SYS_MODULE)
                                return -EACCES;
                        else return 0;
                }
        } else {
                spin_unlock(&selfprot_lock);
                return 0;
        }
}
```

O código acima verifica se a auto-proteção do IMN está ativa. Caso não esteja, permite o acesso. Caso contrário, libera acesso somente para módulos imunológicos. Para outros processos, o tratador checa se ação que o processo deseja executar requer alguma das capacidades exclusivas de processos imunológico, negando o acesso em caso positivo e autorizando em caso contrário.

Ganchos de operações sobre sistemas de arquivos

Completando a implementação do mecanismo de auto-proteção do IMN, são implementados tratadores para os ganchos LSM `security_sb_mount()` e `security_sb_umount()`, os quais interceptam solicitações feitas em espaço de usuário através das chamadas `mount()` e `umount()`, respectivamente.

Os tratadores de auto-proteção do IMN para estes ganchos são utilizados na proteção de dois componentes do *framework*: Interface SecurityFS e TCG, sendo este último um componente externo do IMN, apresentado na Seção 4.5.

A proteção destes componentes é feita através da negação do acesso em execuções das chamadas `mount()` e `mount()` por processos não imunológicos com objetivo de montagem ou desmontagem de sistemas de arquivo do tipo `cgroup` ou `securityfs`, sempre que o mecanismo de auto-proteção estiver ativado.

5.4.3 Interface Administrativa

A Interface Administrativa do IMN utiliza o sistema de arquivos virtual SecurityFS, apresentado na Seção 4.3, para disponibilizar as seguintes operações sobre os recursos do *framework*:

- Ganchos multifuncionais:
  - Registro e desregistro de ganchos para processos imunológicos;
  - Registro e desregistro de tratadores para ganchos de processos imunológicos;
- Auto-proteção:
  - Registro e desregistro de processos imunológicos (módulos imunológicos);
  - Registro e desregistro de `inodes` imunológicos (definição de repositórios seguros);
  - Ativação e desativação do recurso de auto-proteção.

No momento de inicialização do sistema operacional, quando o IMN registra-se junto ao LSM, a interface SecurityFS é montada automaticamente em `/sys/kernel/security/imn` e os seguintes arquivos de controle são criados na raiz do sistema de arquivos virtual:

- `immune_tasks`: lista dos PIDs dos módulos imunológicos. Operações de leitura sobre este arquivo recuperam a lista dos PIDs dos processos registrados no IMN. Arquivo regular, somente leitura;

- `hooks`: lista dos ganchos multifuncionais, módulos imunológicos registrados por gancho e seus respectivos modos de execução. Operações de leitura sobre este arquivo recuperam a lista de todos os ganchos disponibilizados pela arquitetura de ganchos multifuncionais do IMN, bem como o PID dos processos registrados por gancho e o seu modo de execução. Arquivo regular, somente leitura;

- `targets`: lista das funções tratadoras (nome e endereço) por gancho. Operações de leitura sobre este arquivo recuperam a lista de tratadores registrados por módulos imunológicos por gancho. Arquivo regular, somente leitura;

- `selfprot`: controle da funcionalidade de auto-proteção do IMN. Operações de leitura indicam o estado do recurso: 0 desabilitado e 1 habilitado. Operações de escrita sobre este arquivo por parte de processos imunológicos habilitam ou desabilitam o recurso.

Além destes arquivos, os diretórios `operations`, `hook_channels` e `saferepo` disponibilizam mais arquivos de controle:

- `operations`:

  - `REG_TASK` e `UNREG_TASK`: arquivos de controle, somente escrita, para registro de módulos imunológicos. A escrita de *strings* representando o PID do módulo nestes arquivos permitem, respectivamente, o registro e desregistro do processo no *framework*. Quando o mecanismo de auto-proteção está habilitado, apenas processos imunológicos podem registrar novos módulos imunológicos;
  - `REG_HOOK` e `UNREG_HOOK`: arquivos de controle, somente escrita, para registro de ganchos multifuncionais para módulos imunológicos. O registro de ganchos para módulos imunológicos é feita através da escrita de *strings* na forma "`<GANCHO> <N>`", onde o `<GANCHO>` representa o identificador do gancho multifuncional[4] e `N` representa o modo de execução: 0, 1 ou 2. Uma *string* válida seria: `ID_MKDIR 0`. O gancho é registrado para o PID do processo que escreveu sobre o arquivo de controle. Somente processos imunológicos podem escrever neste arquivo. Caso o gancho já esteja registrado para outro processo, o erro `-EADDRINUSE` é retornado;
  - `REG_TARGET` e `UNREG_TARGET`: arquivos de controle, somente escrita, para registro de funções tratadoras para ganchos multifuncionais. Os tratadores de ganchos são registrados através da escrita de *strings* na forma

[4] A listagem completa dos ganchos disponibilizados pelo IMN está disponível no Apêndice A. Esta listagem também pode ser obtida através da leitura do arquivo `hooks` do *framework* IMN.

"<GANCHO> <SIMBOLO>", onde SIMBOLO é o nome de uma função tratadora, definida em um LKM imunológico e exportada através da macro EXPORT_SYMBOL(<SIMBOLO>);

- hook_channels:
  - <GANCHO>: cada gancho multifuncional do IMN disponibiliza um arquivo regular de leitura e escrita no diretório hook_channels. Estes arquivos são os canais de troca de informação entre espaço de *kernel* e os módulos imunológicos;
- saferepo:
  - list: operação de leitura sobre este arquivo de controle listam todos os arquivos protegidos pelo mecanismo de auto-proteção do IMN;
  - load: operações de escrita neste arquivo de controle definem uma nova lista de arquivos protegidos. Os elementos da listagem anterior são descartados;
  - append: operações de escrita neste arquivo de controle adionam arquivos à lista de inodes protegidos;
  - remove: operações de escrita sobre este arquivo de controle remove arquivos da lista de arquivos protegidos, caso o nomes listados existam e estejam presentes na lista;
  - safebin: operações de escrita neste arquivo de controle definem um lista de inodes com permissões de escrita negadas, ao passo que operações de leitura recuperam a lista atual de arquivos protegidos.

5.4.4 Componentes externos

Os *frameworks Netfilter* e *Task Control Group*, apresentados nas Seções 4.4 e 4.5, respectivamente, atendem a uma grande parte dos requisitos do sistema de segurança imunológico, levantados na Seção 5.2. Como estas ferramentas estão em estágio de produção e fazem parte do *kernel* Linux oficial, elas são incluídas integralmente no *framework* IMN, através da proteção destas por meio do recurso de auto-proteção do *framework*.

Além destes componentes, outros componentes externos podem ser incluídos no *framework* IMN, tal como ilustrado na Figura 5.3, através da extensão do mecanismo de auto-proteção por meio de uso de módulos imunológicos e LKMs imunológicos ou definição de repositórios seguros.

Capítulo 6

Análises e testes

No Capítulo 5, os requisitos a serem atendidos pelo *framework* IMN foram detalhados, assim como a arquitetura do *framework* e aspectos de implementação. Na etapa de levantamento de requisitos, Seção 5.2.2, foi feita uma análise dentre os trabalhos desenvolvidos no projeto Imuno visando não somente o levantamento de requisitos, mas também um entendimento dos principais ofensores encontrados no desenvolvimento do sistema de segurança imunológico geral proposto pelo projeto.

Fruto desta análise, passou a ser objetivo do *framework* imunológico não somente o atendimento dos requisitos funcionais do sistema de segurança imunológico em desenvolvimento, mas também características como facilidade de manutenção e atualização da ferramenta. Desta forma, definiu-se uma diretriz a ser seguida na nova implementação do *framework* de segurança imunológica do projeto Imuno: "O *framework* imunológico deve implementar soluções com mínimo impacto no *kernel* Linux e consoantes com as diretrizes e boas práticas adotadas pelos desenvolvedores do Linux".

Dentro desta diretriz, o projeto do *framework* imunológico foi redesenhado e alguns pontos fundamentais foram alterados. O projeto passou a ser implementado utilizando exclusivamente APIs padrão do *kernel* Linux e ferramentas consolidadas do *kernel*, ou seja, não foram feitas alterações no código de qualquer funcionalidade do Linux, tampouco foram incluídos *patches* com ferramentas externas ao *kernel* (não oficiais ou em desenvolvimento) ou novas chamadas de sistema.

Neste capítulo, serão feitas análises e testes sobre a ferramenta desenvolvida. Na Seção 6.1, será avaliada a cobertura dos requisitos levantados na Seção 5.2.3, onde eventuais omissões por conta do atendimento à diretriz de projeto do *framework* serão apontadas. Na Seção 6.2, será apresentado um pequeno sistema segurança implementado segundo a arquitetura desejada para a solução do projeto Imuno e fazendo uso dos recursos do *framework* IMN. O objetivo desta implementação é fazer um ensaio sobre a arquitetura do sistema de segurança imunológica, além de exercitar importantes aspectos funcionais

do IMN. Por fim, na Seção 6.3, é feita uma simulação de ataque bem sucedido a um sistema utilizando a ferramenta descrita na seção anterior, bem como o comportamento da mesma com relação a aspectos de detecção e resposta do ataque.

6.1 Atendimento de requisitos

6.1.1 Requisitos Funcionais

Os requisitos de funcionalidades em nível de *kernel* a serem atendidos pelo *framework* IMN foram divididos em quatro diferentes grupos de tipo de funcionalidades, a saber: monitoramento, resposta, auto-proteção e administração do sistema.

O grupo de requisitos de auto-proteção deve garantir a proteção de todos os componentes do sistema de segurança imunológica: o *framework*, os módulos imunológicos, os LKMs imunológicos, o repositório de análise forense e outros pontos dos sistema de arquivos, indicados pelos módulos imunológicos. O grupo de requisitos de administração do sistema deve prover uma *interface* de comunicação entre os componentes de *kernel* da solução e os componentes em espaço de usuário (módulos imunológicos), de forma a permitir a manipulação das funcionalidades do *framework*, bem como troca de informação entre espaço de núcleo e usuário. Estes dois grupos de requisitos foram integralmente atendidos na implementação do IMN, conforme detalhamento apresentado nas Seções 5.4.2 e 5.4.3.

Já as funcionalidades do grupo de requisitos de monitoramento de processos devem ser capazes rastrear processos e registrar operações como: criação e execução de processos, inserção e remoção de módulos de *kernel* (LKMs), utilização de mecanismos de comunicação interprocessos (IPC) e operações sobre o sistema de arquivos. Obtenção de imagens de memória, informações sobre processos e monitoramento de tráfego de rede também fazem parte deste grupo de requisitos.

Com a utilização da infra-estrutura de ganchos multifuncionais disponibilizada pelo IMN, um LKM imunológico pode definir de maneira bem simples funções tratadoras para os ganchos LSM pertinentes, possibilitando a interceptação de operações de processos, coleta das informações necessárias para análise por parte de módulos imunológicos e escrita destes dados nos canais de saída de cada gancho. Um módulo imunológico de monitoramento poderá, então, registrar-se nos ganchos necessários, definir suas funções tratadoras para os ganchos e, em seguida, ler os dados escritos nos canais de forma a gerar um repositório com estas informações.

Os requisitos de mecanismos de resposta imunológica, por sua vez, incluem diversas capacidades de controle sobre o comportamento de processos, a saber:

- Controle de recursos com estabelecimento de bloqueios, limites e garantias;

- Bloqueio e desbloqueio de processos em espaço de usuário;
- Finalização de processos invasores;
- Recuperação de alterações feitas no sistema de arquivos durante o ataque de um processo invasor.

O *framework* TCG atende boa parte destes requisitos. Conforme visto na Seção 4.5, este *framework* disponibiliza uma série de subsistemas, cada um com uma característica funcional diferente.

Uma relação direta entre o subsistema do TCG, os requisitos de mecanismos de resposta atendidos e uma possível utilização do recurso em um sistema de segurança imunológica é listada a seguir:

- *Cpusets*: estabelecimento de garantia de recursos de memória e CPU para processos. Um módulo imunológico pode, por exemplo, reservar um CPU e um módulo de memória do sistema exclusivamente para processos imunológicos;
- *CFS Bandwidth Control*: estabelecimento de limites para utilização de tempo de CPU. Permite a módulos imunológicos limitar a capacidade de processamento de um processo suspeito;
- *Device Controller*: bloqueio de acesso a dispositivos do sistema operacional. Um módulo imunológico pode impedir um processo de acessar qualquer dispositivo disponível no sistema operacional, como uma porta USB, por exemplo;
- *Freezer*: bloqueio e desbloqueio de processos de maneira eficaz, sem utilização de sinais, de tal forma que o processo paralisado não tenha conhecimento de que foi bloqueado por um período de tempo. Esta abordagem é considerada superior à utilização dos sinais `SIGSTOP` e `SIGCONT`, uma vez que nesta última abordagem um processo pode mudar seu comportamento após tomar conhecimento de que foi bloqueado;
- *Memory Resource Controller* e *HugeTLB Resource Controller*: permitem o estabelecimento de limites para utilização (em *bytes*) de memória física, *swap* e grandes páginas de memória virtual (*HugeTLB*). Um módulo imunológico pode limitar o consumo de memória RAM por parte de um processo suspeito.

Dentre os requisitos de resposta não atendidos pelo TCG, a finalização de processos pode ser facilmente feita em nível de usuário através do envio do sinal `SIGKILL`, por parte de um módulo imunológico, ao processo invasor.

Já a funcionalidade de recuperação de alterações feitas no sistema de arquivos, por um processo invasor durante um ataque, não é atendida em nível de *kernel* pelo *framework*. Apesar da disponibilidade do protótipo UndoFS para atendimento deste requisito, desenvolvido por Fabrício [22] e já utilizado anteriormente no *framework* Imuno [29], este trabalho não utilizou tal ferramenta, deixando a cargo dos desenvolvedores do sistema imunológico a implementação de uma alternativa, até que uma solução deste tipo seja incluída no Linux.

A atualização do UndoFS para o Linux atual, apesar de ser uma opção para o atendimento deste requisito, não é encorajada devido às dificuldades de atualização e manutenção inerentes da solução, conforme exposto anteriormente. Uma alternativa seria o trabalho [11], o qual implementa, em espaço de usuário, um mecanismo de recuperação de alterações em disco.

Por fim, é importante ressaltar que a implementação de módulos de monitoramento, detecção e resposta, por tratarem-se de soluções particulares, inerentes à pesquisa relacionada ao sistema de segurança imunológico, não são considerados requisitos do *framework* IMN e estão fora do escopo deste trabalho, devendo ser implantados por módulos imunológicos.

O módulo de auto-proteção, contudo, apesar de ser considerado um componente imunológico, é implementado diretamente pelo *framework*, por ser necessidade comum a todos os diferentes protótipos já desenvolvidos pelo projeto Imuno e por ser fundamental no contexto de pesquisa em segurança computacional.

6.1.2 Requisitos de Manutenção

Visando minimizar a carga de manutenção necessária para atualização da ferramenta, conforme visto anteriormente, o IMN foi desenvolvido sob a seguinte diretriz: "O *framework* imunológico deve implementar soluções com mínimo impacto no *kernel* Linux e consoantes com as diretrizes e boas práticas adotadas pelos desenvolvedores do Linux".

Por este motivo, o *framework* não adiciona novas chamadas de sistema ou ferramentas não oficiais (externas). Tampouco foram feitas alterações em qualquer ponto do Linux. O IMN é implementado utilizando somente APIs bem definidas, sem qualquer alteração[1] no código do *kernel*. Desta forma, a prova do tempo, ou seja, experiências de atualização do

[1]Dentro da hierarquia de diretórios do código fonte do Linux, o diretório `linux-src/security/` tem em sua raiz os arquivos-fonte das ferramentas de segurança: LSM, SecurityFS e Linux Capabilities. Neste mesmo diretório, existem alguns subdiretórios, um para cada módulo LSM do Linux. Por este motivo, o *patch* do IMN **adiciona** o diretório `linux-src/security/imn/`, onde ficam localizados todos os arquivos do seu código fonte. Os únicos arquivos do *kernel* **alterados** pelo *patch* do *framework* IMN são `linux-src/security/Kconfig` e `linux-src/security/Makefile`, responsáveis pela inclusão das opções de compilação do *framework* (`CONFIG_SECURITY_IMN`).

framework IMN, são um bom indicador do grau de sucesso do atendimento aos requisitos de manutenção.

O IMN teve seu desenvolvimento iniciado sobre a versão 2.6.31 do *kernel* Linux. Desde então, vem sendo atualizado a cada nova versão do *kernel* e, no momento da escrita deste trabalho, encontra-se em pleno funcionamento sobre a versão 3.8 do Linux. O histórico de manutenção, com poucas intervenções para atualização do sistema, demonstra o sucesso da arquitetura adotada pelo IMN com relação ao quesito manutenção de código.

O maior "salto" entre versões do Linux ocorrido, foi a atualização do IMN do Linux versão 2.6.36 para 3.8, onde foi detectada apenas uma alteração de API no Linux que demandou intervenção no código do *framework*. Neste exemplo, houve uma alteração de API na forma de declaração de variáveis do tipo `spin_lock` e `rw_spin_lock`, o que gerou a necessidade de alteração de duas linhas de código do *framework* para retomá-lo a um estado funcional no Linux 3.8:

```
-       imn_hooks_lock = SPIN_LOCK_UNLOCKED;
-       imn_tasks_lock = RW_LOCK_UNLOCKED;
+       imn_hooks_lock = __SPIN_LOCK_UNLOCKED();
+       imn_tasks_lock = __RW_LOCK_UNLOCKED();
```

Além das alterações de API, quaisquer mudanças entre versões dos arquivos do Linux alterados pelo IMN requerem intervenções manuais de atualização. No exemplo anterior, a adição de novos módulos LSM ao Linux exigiu reposicionamento de linhas incluídas pelo IMN nos arquivos `security/Kconfig` e `security/Makefile`.

6.2 Sistema de Segurança opaco utilizando o *framework* IMN

Nesta seção, será montado um cenário simples de utilização do *framework* IMN. Este cenário utiliza um sistema de segurança simplificado, implementado exclusivamente para fins de exercício e validação da utilização do *framework* IMN na criação de um sistema de segurança imunológico geral. O sistema simplificado é composto por três módulos imunológicos e um LKM imunológico, descritos a seguir:

- `modConsole`: Módulo imunológico de administração do sistema de segurança. Sessão de linha de comandos (*bash*), responsável por tarefas administrativas do sistema junto ao *framework* IMN;
- `modMonitor`: Módulo imunológico gerador de informações. Aplicação em C, responsável pela leitura dos canais de ganchos multifuncionais e geração de base de dados de acesso;

- `modResponse`: Módulo de detecção e resposta. Aplicação em C, responsável pela análise do repositório de informações gerado pelo `modMonitor` e intervenções sobre processos suspeitos;

- `modTargs`: LKM imunológico que implementa alguns tratadores de ganchos multifuncionais utilizados pelo `modMonitor` para coleta de informações sobre o sistema operacional.

Este sistema de segurança simplificado simula um sistema de detecção de intrusão baseado em comportamento, e funciona da seguinte forma: quaisquer ações negadas pelo mecanismo de auto-proteção do *framework* IMN são consideradas como comportamento suspeito. Após a detecção de um limite de ações suspeitas por parte de um processo (*soft limit*), este passa a ser considerado suspeito e o sistema de segurança responde diminuindo a oferta de recursos de CPU ao processo em questão. No caso de detecção de mais ações suspeitas por parte de um processo suspeito (*hard limit*), o mesmo é finalizado pelo módulo de resposta.

Neste cenário, cada módulo exercita diferentes funcionalidades do IMN, conforme ilustrado na Tabela 6.1. A seguir, serão detalhados funcionamento e implementação de cada componente do sistema de segurança. Na Seção 6.3 será apresentada uma simulação de utilização da ferramenta.

Componente	**Funcionalidade**	**Escopo**
`modConsole`	Interface SecurityFS	Ajuste de parâmetros do sistema de segurança e consulta de estado de componentes do *framework*
`modMonitor`	Interface SecurityFS	Leitura de canais de saída de ganchos multifuncionais
	Ganchos Multifuncionais	Utilização de tratadores de monitoramento para alimentação de repositório de informações
	Auto-proteção	Utilização do registro histórico de negações de acesso do mecanismo de auto-proteção para alimentação de repositório de informações
`modResponse`	TCG	Limitação recursos de CPU para processos considerados suspeitos
`modTargs`	Ganchos Multifuncionais	Registro de tratadores de monitoramento

Tabela 6.1: Recursos do *framework* IMN utilizados por cada módulo imunológico de um sistema de segurança simplificado.

6.2.1 `modConsole` - Módulo Administrativo

O módulo imunológico `modConsole` é um processo do programa `bash`, utilizado na configuração do sistema de segurança, através da manipulação da *interface* SecurityFS do IMN. Neste módulo, deverão ser feitos os diversos procedimentos necessários para a configuração inicial do sistema de segurança, bem como algumas consultas sobre o estado dos componentes do *framework* IMN.

As atividades de configuração realizadas com este componente são:

- Registro de processos imunológicos;
- Configuração inicial da hierarquia TCG do sistema de segurança;
- Carregamento de LKM imunológico;
- Registro de ganchos e tratadores;
- Ativação do mecanismo de auto-proteção.

Algumas possíveis rotinas de consulta realizadas por este módulo são:

- Listagem de processos imunológicos registrados;
- Listagem de tratadores de ganchos registrado no sistema;
- Listagem de arquivos protegidos pelo mecanismo de auto-proteção;
- Consulta do estado do mecanismo de auto proteção;
- Leitura manual sobre canais de saída de gancho;
- Escrita manual sobre canais de entrada de ganho.

6.2.2 `modMonitor` - Gerador de informações

O módulo imunológico `modMonitor`, é uma aplicação em espaço de usuário que gera uma *thread* de execução para leitura de cada um dos canais dos ganchos imunológicos e uma única *thread* de escrita, para escrita serial no arquivo de *log* do repositório seguro.

As *threads* de leitura executam um *loop*, realizando operações de leitura sobre o arquivo do seu canal. Apesar da implementação com utilização de espera ociosa, estas *threads* não tendem a utilizar muito tempo CPU. Isto acontece devido ao fato de que a implementação do IMN para operações de leitura sobre arquivos do tipo SecurityFS não retorna um arquivo vazio quando não há dados no *buffer* do canal, mas sim coloca a aplicação que invocou a chamada de leitura sobre o arquivo no estado `TASK_INTERRUPTIBLE`, até que haja algum dado a ser lido no canal. Desta forma, as *threads* de leitura do `modMonitor` tendem a ficar boa parte do tempo dormindo, aguardando por dados a serem lidos. Após

a leitura de um *buffer* de dados do seu respectivo canal, uma *thread* de leitura inclui estes dados ao final de uma lista ligada de *jobs* a serem escritos no arquivo de *log*.

A implementação das *threads* de leitura está ilustrada abaixo:

```
int read_channel(void *arg)
{
   ...
   while (1) {
      channel = fopen(channel[hook], "r");
      if (channel) {
         for (i = 0; fread(&buffer[i], 1, 1, channel) == 1; i++) {}
         buffer[i] = '\0';
         fclose(channel);
         if (add_job(buffer, strlen(buffer)) == 0)
            sem_post(&job_queue_count);
      } else
         fprintf(stderr, "Could not open channel %s.\n", channel[hook]);
   }
}
```

A *thread* de escrita, por sua vez, lê os *jobs* da fila e os escreve serialmente no arquivo de *log*. As operações de leitura e escrita na fila de *jobs* é sincronizada com a utilização de um semáforo. Desta forma, a *thread* de escrita executa em *loop* apenas enquanto houverem dados a serem escritos. Nos momentos em que a fila estiver vazia, a *thread* ficará dormindo. A implementação da *thread* de escrita está ilustrada abaixo:

```
int write_jobs(void *arg)
{
...
   while (1) {
      sem_wait(&job_queue_count);
      if (get_job(buf, 1024) == 0) {
         db = fopen(db_path, "a");
         if (db) {
            fprintf(db, "%s", buf);
            fclose(db);
         }
      }
   }
   return 0;
```

```
}
```

Por padrão, os tratadores de auto-proteção de IMN escrevem nos canais de saída dos seus respectivos ganchos, notificações de acessos negados. Ademais, o LKM imunológico deste sistema de segurança adiciona outros tratadores para registro de atividades do sistema imunológico (`modResponse`) sobre processos suspeitos. Desta forma, o repositório forense esperado é um arquivo de *log* com registro de todas as atividades suspeitas realizadas por processos imunológicos e todas as intervenções sobre processos suspeitos por parte do sistema de segurança.

6.2.3 `modResponse` - Detector de Intrusão e Gerador de Respostas

O terceiro e último módulo imunológico deste sistema de segurança simplificado acumula duas das características desejadas ao sistema de segurança imunológica geral a ser desenvolvido pelo projeto Imuno: detecção e resposta. Trata-se de uma aplicação desenvolvida em C, a qual implementa a lógica de detecção de intrusão do sistema e os seus dois mecanismos de resposta: diminuição da capacidade de processamento e finalização de um processo suspeito.

A detecção é feita através da constante leitura e análise dos dados do repositório forense gerado por `modMonitor`. Qualquer negação de acesso realizada pelo módulo de auto-proteção do IMN é considerada uma atitude suspeita e é contabilizada pelo `modResponse`. Sempre que um dado processo atingir a marca de cinco ocorrências suspeitas, ele é movido para a classe de processos suspeitos, a qual impõe capacidade limitada de utilização de tempo de CPU. Aqueles processos que atingirem a marca de 10 atividades suspeitas, são imediatamente finalizados.

A diminuição do poder de processamento é feito através da criação de um grupo chamado `suspects` em hierarquia TCG com o subsistema *CFS Bandwidth Control* ativado. O grupo `suspects` deve ter o atributo `cpu.cfs_quota_us` configurado para dez por cento do valor do atributo `cpu.cfs_period_us`. Desta forma, garante-se que a soma de processamento de todos os processos deste grupo não ultrapassará 10% da capacidade de processamento total do sistema. A finalização do processo, por sua vez, é feita através do simples envio do sinal `SIGKILL` para o processo suspeito.

6.2.4 `modTargs` - LKM Imunológico

Por fim, o LKM imunológico `modTargs` define duas funções tratadoras, responsáveis por registrar nos canais de saída dos seus respectivos ganchos as atuações de processos imunológicos sobre processos suspeitos. Os ganchos utilizados por este módulo são:

`ID_TASK_KILL` e `ID_FILE_OPEN`. O tradador `imuno_file_open()`, quando registrado no gancho `ID_FILE_OPEN`, escreve no canal de saída do gancho operações de abertura para escrita em arquivos de nome `tasks`, diretório `suspects` e sistema de arquivo do tipo `cgroup`. Sua implementação é apresentada abaixo:

```
int imuno_file_open(va_list args, char *ochan)
{
   ...
   file = va_arg(args, struct file *);
   ...
   if (strcmp(file->f_dentry->d_inode->i_sb->s_type->name, "cgroup") == 0
       && strcmp(file->f_dentry->d_parent->d_name.name, "suspects") == 0
       && strcmp(file->f_dentry->d_name.name, "tasks") == 0)
       snprintf(ochan, 1024, "%s [IMN:modMonitor][%d][%d] [LOG]"
            "[@file_open()]: [fstype %s dname %s]\n",
            date, current->parent->pid, current->pid,
            file->f_dentry->d_inode->i_sb->s_type->name,
            file->f_dentry->d_name.name);
   return 0;
}
```

O tratador `imn_task_kill()`, por sua vez, registra todas as ocorrências da chamada de sistema `kill(SIGKILL)`, e possui a seguinte implementação:

```
int imuno_task_kill(va_list args, char *ochan)
{
   ...
   sig = va_arg(args, int);
   ...
   if (sig == SIGKILL)
      snprintf(ochan, 1024, "%s [IMN:modMonitor][%d][%d] [LOG]"
         "[@task_kill()]: [pid %d signal SIGKILL]\n",
         timestamp, current->parent->pid, current->pid, p->pid);
   return 0;
}
```

Os registros abaixo ilustram possíveis saídas destes tratadores:

```
[IMN:modMonitor][701][702] [LOG][@file_open()]: [fstype cgroup dname tasks]
[IMN:modMonitor][455][466] [LOG][@task_kill()]: [pid 569 signal SIGKILL]
```

6.3 Simulação de uso

Na seção anterior, foi apresentado o funcionamento dos componentes integrantes de um pseudo sistema de segurança imunológico. Nesta seção, será apresentada uma simulação de ataque sobre este sistema, bem como o comportamento de cada um dos seus componentes.

Setup Inicial

Inicialmente, os três módulos imunológicos do sistema de segurança devem ser executados. Uma configuração inicial deve então ser feita pelo `modConsole`. Este módulo deve primeiramente registrar os processos imunológicos: ele próprio e os módulos `modMonitor` e `modResponse`, que possuem os PIDs 405, 459 e 505, respectivamente:

```
# cd /var/sys/kernel/security/imn
# echo $$ > operations/REG_TASK
# echo 459 > operations/REG_TASK
# echo 505 > operations/REG_TASK
# cat immune_tasks
405
459
505
```

A seguir, deve-se registrar os ganchos multifuncionais e tratadores de ganchos a serem utilizados pelo módulo `modMonitor`.

- Registro de ganchos:

```
# echo ID_FILE_OPEN 0 > operations/REG_HOOK
# echo ID_TASK_KILL 0 > operations/REG_HOOK
# cat hooks | grep $$
#    Hook_Name           PID   MODEXEC   NTARGS
43   ID_FILE_OPEN        405   0         0
63   ID_TASK_KILL        405   0         1
```

- Registro de tratadores:

```
# insmod ~/modTargs.ko
# cat targets
#Hook   #Target Target_Name
  43       0    imuno_file_open+0x0/0x140 [targs]
```

```
  63       0    imn_selfprot_task_kill+0x0/0xc0
  63       1    imuno_task_kill+0x0/0x100 [targs]
# cat hooks | grep $$
#    Hook_Name           PID   MODEXEC   NTARGS
43   ID_FILE_OPEN        416   0         1
63   ID_TASK_KILL        416   0         2
```

Os passos finais da inicialização do sistema são a configuração do repositório seguro do mecanismo de auto-proteção do IMN e ajustes do *framework* TCG de acordo com as necessidades do sistema de segurança.

- Definição de arquivos do repositório seguro:

```
# mkdir -p /imuno/saferepo
# touch /imuno/saferepo/forensic.db
# echo -e '/imuno/saferepo/forensic.db\n' > saferepo/load
# cat saferepo/list
/srv/saferepo/forensic.db
```

- Configuração do *framework* TCG:

```
# mkdir /imuno/tcg
# mount -t cgroup -o freezer,cpu cgroup /imuno/tcg/
# mkdir /imuno/tcg/suspects
# echo 100000 > /imuno/tcg/suspects/cpu.cfs_period_us
# echo 10000 > /imuno/tcg/suspects/cpu.cfs_quota_us
```

Finalmente o sistema de segurança pode ser ativado, a partir da habilitação do mecanismo de auto-proteção do *framework* IMN:

```
# echo 1 > selfprot
```

Simulação de Ataque

Neste ponto, considera-se que houve um ataque bem sucedido no sistema operacional e um processo malicioso conseguiu iniciar uma sessão de *bash* como usuário *root*. O atacante inicia então uma série de comandos visando desabilitar os mecanismos de proteção do sistema e os registros de suas atividades, conforme abaixo:

```
# cat /imuno/tcg/suspects/tasks
cat: /imuno/tcg/suspects/tasks: Operation not permitted
# umount /imuno/tcg
umount: /imuno/tcg: must be superuser to umount
# mount -t securityfs securityfs /tmp/
mount: permission denied
# insmod /tmp/targs.ko
insmod: error inserting '/tmp/targs.ko': -1 Operation not permitted
# cat /imuno/saferepo/forensic.db
cat: /imuno/saferepo/forensic.db: Operation not permitted
# echo "" > /imuno/saferepo/forensic.db
-su: /imuno/saferepo/forensic.db: Operation not permitted
# iptables -L
iptables v1.4.4: can't initialize iptables table `filter':
   Permission denied (you must be root)
# kill -9 405
-su: kill: (405) - Operation not permitted
# kill -9 459
-su: kill: (459) - Operation not permitted
# kill -9 505
-su: kill: (505) - Operation not permitted
Connection to imn closed.
```

A negação de acesso para todos os comandos acima é decorrente do exercício de diferentes tratadores de ganchos multifuncionais do mecanismo de auto-proteção do IMN, conforme visto na Seção 5.4.2. A cada negação de acesso, o tratador de auto-proteção registra no seu respectivo canal de gancho alguns dados da ocorrência. No sistema de segurança implementado, o módulo `modMonitor` captura estes registros e os escreve no arquivo `/imuno/selfprot/forensic.db`. Para a sequencia de comandos acima, o seguinte arquivo de *log* é gerado pelo módulo imunológico:

```
[IMN:Selfprot][574][590] [DENIED][@inode_permission()]: [fstype cgroup]
[IMN:Selfprot][574][596] [DENIED][@sb_umount()]: [fstype cgroup]
[IMN:Selfprot][574][612] [DENIED][@sb_mount()]: [fstype securityfs]
[IMN:Selfprot][574][631] [DENIED][@capable()]: [CAP_SYS_MODULE]
[IMN:Selfprot][574][662] [DENIED][@inode_permission()]: [inode df2a98f0]
[IMN:modMonitor][504][505] [LOG][@file_open()]:[fstype cgroup dname task]
[IMN:Selfprot][573][574] [DENIED][@inode_permission()]: [inode df2a98f0]
[IMN:Selfprot][574][691] [DENIED][@capable()]: [CAP_NET_ADMIN]
```

```
[IMN:Selfprot][573][574] [DENIED][@task_kill()]: [pid 405]
[IMN:Selfprot][573][574] [DENIED][@task_kill()]: [pid 459]
[IMN:Selfprot][573][574] [DENIED][@task_kill()]: [pid 505]
[IMN:modMonitor][504][505] [LOG][@task_kill()]: [pid 574 signal SIGKILL]
```

As duas linhas com prefixo `[IMN:modMonitor]` na listagem acima indicam as ações realizadas pelo módulo `modResponse`, capturadas pelo LKM `modTargs` e escritas no arquivo de *log* pelo módulo `modMonitor`: inicialmente, o processo suspeito tem seu poder de processamento limitado através de sua inclusão no grupo `suspects` do *framework* TCG. Em seguida, como o o módulo imunológico constata que trata-se de um ataque e finaliza o processo suspeito através da chama de sistema `kill(SIGKILL)`.

Capítulo 7

Conclusões

Ao longo do seu desenvolvimento, este trabalho sofreu algumas alterações de escopo. O planejamento inicial era a atualização de uma ferramenta de *software* desenvolvida pelo projeto Imuno e ampliação do seu rol de funcionalidades, através do levantamento de novos requisitos funcionais junto aos pesquisadores integrantes do grupo de pesquisa.

A primeira mudança ocorreu ainda nas etapas iniciais deste trabalho. No Capítulo 4, foram apresentadas diversas ferramentas de segurança existentes na versão oficial do *kernel* do sistema operacional GNU/Linux. Diferentemente do cenário existente no momento das implementações de protótipos do projeto Imuno anteriores a este trabalho, onde grande parte do ferramental utilizado estava em fase de desenvolvimento ou mesmo foi implementado pelo grupo, devido à não existência de soluções equivalentes, as ferramentas apresentadas naquele capítulo são consolidadas, estáveis e oferecem boa cobertura dos requisitos necessários ao sistema de segurança imunológico, levantados na Seção 5.2, sendo, portanto, utilizadas em conjunto com as funcionalidades implementadas pelo IMN.

Ademais, a revisão de conceitos fundamentais do funcionamento do *kernel* Linux, apresentados no Capítulo 3, somada às experiências acumuladas nos trabalhos de Marcelo, Diego, Fabrício e Martim, apresentados nas Seções 2.2 e 5.2, levaram à conclusão de que, soluções implementadas em nível de *kernel*, e que alterem profundamente o seu código, possuem uma manutenção muito onerosa, devendo ser evitadas sempre que possível.

Como o objetivo do final deste trabalho era disponibilizar uma ferramenta funcional e de manutenção tão simples quanto possível, a arquitetura do *framework* imunológico foi refeita e em seguida implementada através ferramenta batizada de IMN.

Chegara então a hora da execução de outra etapa previsa na proposta inicial deste trabalho: levantamento de requisitos de novas funcionalidades necessárias ao *framework* imunológico. Naquele momento, contudo, o projeto Imuno passou por mudanças profundas em sua estrutura. O foco da pesquisa foi direcionado para outras áreas de segurança de sistemas e o objetivo de construção de um sistema de segurança imunológico geral foi

temporariamente adiado.

Apesar de parcialmente afetado com estas mudanças, considerando toda a pesquisa sobre o *kernel* Linux e suas questões relativas à controle de acesso, controle de recursos e segurança em geral, optou-se pela continuidade do desenvolvimento deste trabalho, por dois principais motivos: primeiramente, já havia ampla disponibilidade de requisitos necessários para uma solução imunológica; depois, a generalidade da solução que estava sendo concebida poderia ser utilizada como plataforma para o desenvolvimento de outras soluções de segurança, não necessariamente relacionadas com os princípios de funcionamento do sistema imunológico humano.

A partir dos resultados apresentados no Capítulo 6, pode-se afirmar que o *framework* IMN cumpre sua proposta inicial de entregar aos seus usuários, desenvolvedores de módulos de segurança inspirados no sistema imunológico humano, um pacote de funcionalidades com flexibilidade e simplicidade de uso e manutenção, reduzindo a complexidade de implementação do sistema proposto. Cumpre também o objetivo de ser uma ferramenta a ser utilizada como base comum para implementação do sistema de segurança proposto, diminuindo, portanto, o retrabalho inerente à implementação de requisitos comuns, como auto-proteção, no desenvolvimento de diferentes módulos do sistema de segurança imunológica.

Desta forma, este trabalho traz as seguintes contribuições ao projeto Imuno e à comunidade científica:

- Farta documentação sobre as ferramentas de segurança atualmente disponíveis no *kernel* Linux; certamente um importante produto deste trabalho, e que poderá ser de grande utilidade para os integrantes do projeto e outros pesquisadores interessados no funcionamento dos aspectos de segurança do Linux e na implementação de ferramentas relacionadas;

- Diminuição significativa do ônus de manutenção e atualização do *framework* imunológico a ser utilizado como plataforma para o desenvolvimento de um sistema de segurança imunológico geral, aproximando ainda mais essa peça de *software* de uma versão com maturidade suficiente para pleitear uma admissão na versão oficial do Linux;

- Nova arquitetura de ganchos multifuncionais, a qual permite que funções tratadoras para ganchos LSM possam ser implementadas na forma de LKMs. Esta implementação, além de ter sua utilidade no escopo do projeto Imuno, pode ser utilizada para acelerar o processo de desenvolvimento e depuração de sistemas de segurança que façam uso do *framework* LSM, uma vez que, a partir da versão 2.6.24 do Linux, um módulo LSM não mais pode ser definido como um LKM, o que dificulta

o processo de desenvolvimento, uma vez que exige uma compilação geral do *kernel* e reinicialização do sistema operacional a cada pequena alteração no código do módulo LSM em desenvolvimento. Portanto, utilizar a característica modular do IMN durante o processo de desenvolvimento de um módulo LSM certamente tornaria o processo mais célere. A generalidade da solução desenvolvida permite ainda o desenvolvimento de outros sistemas de segurança, não necessariamente inspirados no sistema imunológico humano.

- Implementação de um controle de acesso mandatório simplificado, o mecanismo de auto-proteção, que, assim como o mecanismo de ganchos multifuncionais com tratadores modulares, além de atender às necessidades do sistema de de segurança do projeto Imuno, pode ser utilizado para outros fins, reforçando ainda mais a segurança do sistema operacional GNU/Linux.

Dentre os trabalhos futuros que o autor pretende desenvolver a partir dos resultados obtidos neste trabalho, estão:

- Extensão do mecanismo de auto-proteção, de forma a permitir o estabelecimento políticas de controle de acesso mandatório mais flexíveis, de uma forma similar àquela desenvolvida pelo LSM SMACK, apresentado na Seção 4.2.3. Na implementação atual, um administrador de serviços de rede, por exemplo, deve registrar um *shell* imunológico para proceder com algumas tarefas administrativas, tal como o ajuste de regras de *firewall* do sistema. Outra alternativa seria desabilitar a auto-proteção durante a execução de rotinas administrativas. Ambas abordagens são extremas e podem ser evitadas através da implementação de políticas de controle de acesso mais flexíveis;

- Desenvolvimento de um módulo de controle de acesso capaz de aplicar controle de acesso mandatório sobre credenciais Linux do tipo *capabilities*. Tal módulo deverá permitir ao administrador estabelecer grupos de processos ou usuários e definir quais as capacidades devem ser atribuídas às credenciais daquele grupo. Atualmente, o controle de acesso sobre este tipo de credencial é feito de maneira discricionária, uma vez que as credenciais de objetos são armazenadas em *inodes* e herdadas por processos que tenham permissões discricionárias (propriedade) sobre tais arquivos;

- Extensão do *framework* de forma a permitir um empilhamento de diferentes módulos LSM. Conforme apresentado no Capítulo 4, *kernel* Linux possui diversos módulos LSM, todos com características específicas. Estes módulo, contudo, não podem coexistir, pois utilizam a mesma estrutura de ganchos LSM. O *framework* IMN, por sua vez, permite o registro de vários tratadores diferentes para um mesmo

gancho LSM. Portanto, o empilhamento de módulos LSM através da utilização do mecanismo de ganchos multifuncionais do IMN é, em tese, possível;

- Implementação eficiente de sistema de arquivos com funcionalidades de recuperação de dados com as características do *middleware* UNDOFS.

Referências Bibliográficas

[1] Linux programmer's manual: ptrace. [Online; acessado em 16 de fevereiro de 2013].

[2] Linux programmer's manual: syscalls. [Online; acessado em 16 de março de 2013].

[3] Overview of standard C libraries on Linux. [Online; acessado em 16 de março de 2013].

[4] GNU General Public License. [Online; acessado em 20 de março de 2013].

[5] Why GRSecurity cannot use LSM. [Online; acessado em 25 de janeiro de 2013].

[6] Josh Aas. Understanding the Linux 2.6.8.1 CPU scheduler. *Retrieved Oct*, 16, 2005.

[7] R. Anderson and A. Khattak. The use of information retrieval techniques for intrusion detection. In *Proceedings of First International Workshop on the Recent Advances in Intrusion Detection (RAID)*, 1998.

[8] Michael Bacarella. Taking advantage of Linux capabilities. *Linux Journal*, 2002.

[9] Paolo Baldan, Andrea Corradini, Javier Esparza, Tobias Heindel, Barbara König, and Vitali Kozioura. Verifying red-black trees. In *Proc. of COSMICAH*, 2005.

[10] Mark Burgess. NSA National Information Assurance Reserach Laboratory. [Online; acessado em 10 de fevereiro de 2013].

[11] Gabriel Dieterich Cavalcante. *Detecção e Recuperação de Intrusão com uso de Controle de Versão*. Dissertação de Mestrado, Universidade Estadual de Campinas, 2010.

[12] Kees Cook. Yama documentation. [Online; acessado em 16 de fevereiro de 2013].

[13] Jonathan Corbet. SecurityFS. [Online; acessado em 2 de fevereiro de junho de 2013].

[14] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. Subdomain: Parsimonious server security. In *USENIX 14th Systems Administration Conference (LISA)*, 2000.

[15] André Augusto da Silva Pereira. Implementação de controle de recursos e extensão de funcionalidades de framework de seguranaça imunológica. Technical report, Proposta de Mestrado, Universidade Estadual de Campinas, 2009.

[16] P Bovet Daniel and Cesati Marco. Understanding the Linux kernel. *Sebastopol, CA, US, O'Reilly*, 2005.

[17] Dipankar Dasgupta. Immunity-based Intrusion Detection System: a General Framework. In *Proceedings of the 22nd NISSC*, pages 147–160, 1999.

[18] Dipankar Dasgupta and Hal Brian. Mobile Security Agents for Network Traffic Analysis. In *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*. IEEE, 2001.

[19] Dipankar Dasgupta and Fabio González. An Immunity-based Technique to Characterize Intrusions in Computer Networks. *Evolutionary Computation, IEEE Transactions on*, pages 281–291, 2002.

[20] Diego Assis de Monteiro Fernandes. *Resposta Automática em um Sistema de Segurança Imunológico Computacional*. Dissertação de Mestrado, Universidade Estadual de Campinas, 2003.

[21] Fabricio de Paula and Paulo Licio de Geus. Attack evidence detection, recovery, and signature extraction with ADENOIDS. *Telecommunications and Networking-ICT 2004*, pages 1083–1092, 2004.

[22] Fabrício Sérgio de Paula. *Uma Arquitetura de Segurança Computacional Inspirada no Sistema Imunológico*. Tese de Doutorado, Universidade Estadual de Campinas, 2004.

[23] Fabrício Sérgio de Paula, Marcelo Abdalla dos Reis, Diego Fernandes, and Paulo Lício de Geus. Modelagem de um Sistema de Segurança Imunológico. *Anais do III Simpósio de Segurança em Informática*, pages 91–100, 2001.

[24] Fabrício Sérgio de Paula, Marcelo Abdalla dos Reis, Diego Fernandes, and Paulo Lício de Geus. ADENOIDS: A Hybrid IDS Based on the Immune System. *Proceedings of the 9th International Conference on Neural Information Processing*, 2002.

[25] Fabrício Sérgio de Paula, Marcelo Abdalla dos Reis, Diego Fernandes, and Paulo Lício de Geus. A Hybrid IDS Architecture Based on the Immune System. *Anais do II Workshop Brasileiro de Segurança de Sistemas Computacionais*, pages 33–40, 2002.

[26] F.S. de Paula, L.N. de Castro, and P.L. de Geus. An intrusion detection system using ideas from the immune system. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 1, pages 1059–1066. IEEE, 2004.

[27] Patrik D'haeseleer, Stephanie Forrest, and Paul Helman. An Immunological Approach to Change Detection: Algorithms, Analysis and Implications. *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*, 1996.

[28] Patrik D'haeseleer, Stephanie Forrest, and Paul Helman. A Distributed Apporach to Anomaly Detection. *ACM Transactions on Information System Security*, 1997.

[29] Martim d'Orey Posser de Andrade Carbone. *Framework de* K*ernel para um* S*istema de* S*egurança* I*munológico*. Dissertação de Mestrado, Universidade Estadual de Campinas, 2006.

[30] Marcelo Abdalla dos Reis. *Forense* C*omputacional e sua* A*plicação em* S*egurança* I*munológica*. Dissertação de Mestrado, Universidade Estadual de Campinas, 2003.

[31] Fernando Esponda, Elena Ackley, Stephanie Forrest, and Paul Helman. On-line Negative Databases. *Proceedings of the 3rd International Conference on Artificial Immune Systems*, 2004.

[32] Michael Fox, John Giordano, Lori Stotler, and Arun Thomas. SELinux and GRSecurity: a case study comparing Linux security kernel enhancements. 2009.

[33] Simson Garfinkel, Gene Spafford, and Alan Schwartz. *Practical* U*nix & I*nternet S*ecurity, 3rd* E*dition*. O'Reilly Media, Inc., 2003.

[34] Matthew Glickman, Justin Balthrop, and Stephanie Forrest. A Machine Learning Evaluation of an Artificial Immune System. *Evolutionary Computation*, pages 179–212, 2005.

[35] S.E. Hallyn and A.G. Morgan. Linux capabilities: making them work. In *Linux Symposium*, page 163, 2008.

[36] Serge Edward Hallyn and Phil Kearns. *Domain and* T*ype* E*nforcement in* L*inux*. PhD thesis, College of William and Mary, 2003.

[37] T Harada, T Horie, and K Tanaka. Towards a manageable Linux security. In *Linux Conference*, volume 2005, 2005.

[38] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. Access policy generation system based on process execution history. *Network Security Forum*, 2003.

[39] Steven Hofmyer and Stephanie Forrest. Immunity by design: An Artificial Immune System. *Proceedings of the Genetic and Evolutionary Computation Conference*, 1999.

[40] David Howells. Credentiasl in Linux. [Online; acessado em 13 de fevereiro de 2013].

[41] Takamasa Isohara, Keisuke Takemori, Yutaka Miyake, Ning Qu, and Adrian Perrig. LSM-based secure system monitoring using kernel protection schemes. In *Availability, Reliability, and Security, 2010. ARES'10 International Conference on*, pages 591–596. IEEE, 2010.

[42] Jeffrey O. Kephart. A Biologically Inspired Immune System for Computers. *Artificial Life IV: Proceedings of the 4th International Workshop on the Synthesis and Simulation of Living Systems*, pages 130–139, 1994.

[43] R. Krishnakumar. HugeTLB — Large Page Support in the Linux Kernel. *Linux Gazette*, 155, 2008.

[44] J. Lepreau, R. Spencer, S. Smalley, P. Loscocco, M. Hibler, and D. Andersen. The Flask security architecture: System suporte for diverse security policies. *Proceedings of the Eighth USENIX Security Symposium*, 1999.

[45] A.G. Litke. "turning the page" on hugetlb interfaces. In *Linux Symposium*, page 277, 2007.

[46] Peter Loscocco and Stephen D Smalley. Meeting critical security objectives with Security-Enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, pages 115–134, 2001.

[47] Paul Menage. Control Groups definition, implementation details, examples and API. [Online; acessado em 17 de julho de 2009].

[48] Paul B Menage. Adding Generic Process Containers to the Linux Kernel. *Proceedings of the Linux Symposium*, 2007.

[49] Ingo Molnar. Modular Scheduler Core and Completely Fair Scheduler. [Online; acessado em 2 de março de 2013].

[50] Shailabh Nagar, Humbertur Franke, Jonghyuk Choi, Chandra Seetharaman, Sott Kaplan, Nivedita Singhvi, Vivek Kashyap, and Mike Kravetz. Class-based Prioritized Resource Control in Linux. *Proceedings of the 2003 Ottawa Linux Symposium*, pages 161–180, 2003.

[51] United States Government Department of Defense. DoDD 5200.28-STD. Trusted Computer System Evaluation Criteria, 1986.

[52] U.S. Department of Health and Human Services. *Understanding the* I*mmune* S*ystem*. National Institutes of Health, 2003.

[53] A. Ott and S. Fischer-Hubner. The 'Rule Set Based Access Control'(RSBAC) framework for Linux. In *Proceedings of the 8th International Linux Kongress*, 2001.

[54] Amon Ott. Why RSBAC does not use LSM. [Online; acessado em 25 de janeiro de 2013].

[55] NSA Peter Loscocco. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA*, page 29. USENIX Association, 2001.

[56] Love Robert. *Linux kernel development*. Pearson Education India, 2004.

[57] Casey Schaufler. Smack in embedded computing. In *Proceedings of the 10th Linux Symposium*, 2008.

[58] Stephen Smalley and Timothy Fraser. A security policy configuration for the Security-Enhanced Linux. Technical report, NAI Labs Technical Report, 2001.

[59] Anil Somayaji. *Operating* S*ystem* S*tability and* S*ecurity through* P*rocess* H*omeostasis*. Tese de Doutorado, University of New Mexico, 2002.

[60] Anil Somayaji and Stephanie Forrest. Automated Response Using System-call Delays. *Proceedings of the 9th USENIX Security Symposium*, 2000.

[61] Andrew S Tanenbaum. *Modern operating systems*, volume 2. Prentice Hall New Jersey, 1992.

[62] P Turner, BB Rao, and N Rao. CPU bandwidth control for CFS. *Proceedings of the Ottawa Linux Symposium*, 2010.

[63] Chee Siang Wong, Ian Tan, Rosalind Deena Kumari, and Fun Wey. Towards achieving fairness in the Linux scheduler. *ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel*, 2008.

[64] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. *Proceedings of the 11th USENIX Security Symposium*, 2002.

Apêndice A

Ganchos Multifuncionais

O mecanismo de ganchos multifuncionais apresentado na Seção 5.4.1 disponibiliza uma *interface* para registro de funções tratadoras definidas em LKMs (*Linux kernel modules)* para fins de monitoramento ou implementação de mecanismos de controle de acesso particulares.

As funções tratadoras devem ter assinaturas, de acordo com o tipo de retorno esperado pelo gancho LSM, da seguinte forma:

`int nome(va_list args, char *ochan);`

ou

`void nome(va_list args, char *ochan);`

A lista de argumentos `args` varia de acordo com o gancho utilizado. Os parâmetros são recuperados pela função `va_arg()` na ordem em que são definidos na assinatura do gancho LSM original.

A função abaixo ilustra a definição de um tratador para o grancho multifuncional `ID_MKDIR`, incluindo a recuperação dos parâmetros originais do gancho:

```
int target_for_mkdir(va_list args, char *ochan)
{
        struct inode *dir;
        struct dentry *dentry;
        int mode;

        dir = va_arg(args, struct inode *);
        dentry = va_arg(args, struct dentry *);
        mode = va_arg(args, int);

        sprintf(ochan, "PID %d criou o diretório %s.\n",
             current->pid, dentry->d_name.name");
```

```
        return -1;
}
```

A seguir, serão listados todos os ganchos LSM disponíveis na versão 3.8 do Linux e disponibilizados pelo mecanismo de ganchos multifuncionais do *framework* IMN.

Ganchos em operações de execução de programas

Gancho	Descrição	Parâmetros
ID_BPRM_SET_CREDS	Intercepta a função `prepare_binprm()`, responsável pelo preenchimento de estruturas `linux_binprm` a partir das credenciais do processo que invocou o processo e as credenciais do `inodes` binário que está sendo executado pela chamada `execve()`	`struct linux_binprm *bprm`
ID_BPRM_CHECK_SECURITY	Intercepta a função `search_binary_handler()`, responsável pela busca do tratador adequado para o binário correspondente ao `inode` que está sendo executado. Neste gancho, as variáveis de ambiente do processo `argv` e `envp` podem ser acessádas de maneira confiável	`struct linux_binprm *bprm`
ID_BPRM_COMMITTING_CREDS	Intercepta a função `install_exec_creds()`, responsável por instalar no novo processo as credenciais definidas pela função `prepare_binprm`	`struct linux_binprm *bprm`
ID_BPRM_COMMITTED_CREDS	Intercepta a mesma função da chamada anterior, neste caso, logo após a chamada `commit_creds(bprm->cred);`	`struct linux_binprm *bprm`
ID_BPRM_SECUREEXEC	Intercepta a função `create_elf_tables()`, que retorna um valor booleano, verdadeiro ou falso, indicando se a `libc` deve utilizar ou não o seu modo seguro	`struct linux_binprm *bprm`

Ganchos em operações sobre sistemas de arquivos

Gancho	Função interceptada	Parâmetros
ID_SB_ALLOC_SECURITY	`alloc_super()`	`struct super_block *sb`
ID_SB_FREE_SECURITY	`destroy_super()`	`struct super_block *sb`
ID_SB_STATFS	`statfs_by_dentry()`	`struct dentry *dentry`
ID_SB_COPY_DATA	`mount_fs()`	`char *orig`
		`char *copy`

Gancho	Função interceptada	Parâmetros
ID_SB_MOUNT	do_mount()	const char *dev_name
		struct path *path
		const char *type
		unsigned long flags
		void *data
ID_SB_REMOUNT	do_remount()	struct super_block *sb
		void *data
ID_SB_UMOUNT	do_umount()	struct vfsmount *mnt
		int flags
ID_SB_PIVOTROOT	SYSCALL_DEFINE2(pivot_root, ...)	struct path *old_path
		struct path *new_path
ID_SB_SET_MNT_OPTS	nfs_set_sb_security()	struct seq_file *m
		struct super_block *sb
ID_SB_CLONE_MNT_OPTS	nfs_clone_sb_security()	struct super_block *oldsb
		struct super_block *newsb
ID_SB_PARSE_OPTS_STR	nfs_parse_mount_options()	char *options
		struct sec_mnt_opts *opts

Ganchos em operações sobre estruturas do tipo `inode`

Gancho	Função interceptada	Parâmetros
ID_INODE_ALLOC_SECURITY	inode_init_always()	struct inode *inode
ID_INODE_FREE_SECURITY	__destroy_inode()	struct inode *inode
ID_INODE_INIT_SECURITY	btrfs_xattr_security_init()	struct inode *inode
	ext2_init_security()	struct inode *dir
	ext3_init_security()	const struct qstr *qstr
	ext4_init_security()	char **name
	gfs2_security_init()	void **value
	...	size_t *len
ID_INODE_CREATE	vfs_create()	struct inode *dir
		struct dentry *dentry
		umode_t mode
ID_INODE_LINK	vfs_link()	struct dentry *old_dentry
		struct inode *dir
		struct dentry *new_dentry
ID_INODE_UNLINK	vfs_unlink()	struct inode *dir
		struct dentry *dentry
ID_INODE_SYMLINK	vfs_symlink()	struct inode *dir
		struct dentry *dentry
		const char *old_name
ID_INODE_MKDIR	vfs_mkdir()	struct inode *dir
		struct dentry *dentry
		umode_t mode

Gancho	Função interceptada	Parâmetros
ID_INODE_RMDIR	vfs_rmdir()	struct inode *dir
		struct dentry *dentry
ID_INODE_PERMISSION	__inode_permission()	struct inode *inode
		int mask
ID_INODE_MKNOD	vfs_mknod()	struct inode *dir
		struct dentry *dentry
		umode_t mode
		dev_t dev
ID_INODE_RENAME	vfs_rename_dir()	struct inode *old_dir
	vfs_rename_other()	struct dentry *old_dentry
		struct inode *new_dir
		struct dentry *new_dentry
ID_INODE_READLINK	SYSCALL_DEFINE4(readlinkat, ...)	struct dentry *dentry
ID_INODE_FOLLOW_LINK	follow_link()	struct dentry *dentry
		struct nameidata *nd
ID_INODE_SETATTR	vfs_getattr()	struct dentry *dentry
		struct iattr *attr
ID_INODE_GETATTR	notify_change()	struct vfsmount *mnt
	fat_ioctl_set_attribute()	struct dentry *dentry
ID_INODE_SETXATTR	vfs_setxattr()	struct dentry *dentry
		const char *name
		const void *value
		size_t size
		int flags
ID_INODE_POST_SETXATTR	__vfs_setxattr_noperm()	struct dentry *dentry
		const char *name
		const void *value
		size_t size
		int flags
ID_INODE_GETXATTR	vfs_getxatt()	struct dentry *dentry
		const char *name
ID_INODE_LISTXATTR	vfs_listxattr()	struct dentry *dentry
ID_INODE_REMOVEXATTR	vfs_removexattr()	struct dentry *dentry
		const char *name
ID_INODE_GETSECURITY	xattr_getsecurity()	const struct inode *inode
		const char *name
		void **buffer
		bool alloc
ID_INODE_SETSECURITY	sysfs_setxattr()	struct inode *inode
	__vfs_setxattr_noperm()	const char *name
		const void *value
		size_t size
		int flags

Gancho	Função interceptada	Parâmetros
ID_INODE_LISTSECURITY	vfs_listxattr() sockfs_listxattr()	struct inode *inode
		char *buffer
		size_t buffer_size
ID_INODE_NEED_KILLPRIV	notify_change() file_remove_suid()	struct dentry *dentry
ID_INODE_KILLPRIV	notify_change() file_remove_suid()	struct dentry *dentry
ID_INODE_GETSECID	audit_copy_inode() ima_match_rules()	const struct inode *inode
		u32 *secid
ID_PATH_MKNOD	SYSCALL_DEFINE4(mknodat, ...)	struct path *dir
		struct dentry *dentry
		umode_t mode
		unsigned int dev
ID_PATH_TRUNCATE	handle_truncate() do_sys_truncate() do_sys_ftruncate()	kgid_t gid
ID_PATH_UNLINK	do_unlinkat() cachefiles_bury_object()	struct path *dir
		struct dentry *dentry
ID_PATH_SYMLINK	SYSCALL_DEFINE3(symlinkat, ...)	struct path *dir
		struct dentry *dentry
		const char *old_name
ID_PATH_LINK	SYSCALL_DEFINE5(linkat, ...)	struct dentry *old_dentry
		struct path *new_dir
		struct dentry *new_dentry
ID_PATH_MKDIR	SYSCALL_DEFINE3(mkdirat, ...)	struct path *dir
		struct dentry *dentry
		umode_t mode
ID_PATH_RMDIR	do_rmdir()	struct path *dir
		struct dentry *dentry
ID_PATH_RENAME	SYSCALL_DEFINE4(renameat, ...)	struct path *old_dir
		struct dentry *old_dentry
		struct path *new_dir
		struct dentry *new_dentry
ID_PATH_CHMOD	chmod_common()	struct path *path
		umode_t mode
ID_PATH_CHOWN	chown_common()	struct path *path
		kuid_t uid
		kgid_t gid
ID_PATH_CHROOT	SYSCALL_DEFINE1(chroot, ...)	kgid_t gid

Ganchos em operações sobre estruturas do tipo `file`

Gancho	Função interceptada	Parâmetros
ID_FILE_PERMISSION	do_fallocate() rw_verify_area() vfs_readdir()	struct file *file int mask
ID_FILE_ALLOC_SECURITY	get_empty_filp()	struct file *file
ID_FILE_FREE_SECURITY	__fput() put_filp()	struct file *file
ID_FILE_IOCTL	SYSCALL_DEFINE3(ioctl, ...)	struct file *file unsigned int cmd unsigned long arg
ID_MMAP_ADDR	get_unmapped_area() expand_downwards() validate_mmap_reques()	unsigned long addr
ID_MMAP_FILE	do_shmat() vm_mmap_pgoff()	struct file *file unsigned long reqprot unsigned long prot unsigned long flags
ID_FILE_MPROTECT	SYSCALL_DEFINE3(mprotect, ...)	struct vm_area_struct *vma unsigned long reqprot unsigned long prot
ID_FILE_LOCK	SYSCALL_DEFINE2(flock, ...)	struct file *file unsigned int cmd
ID_FILE_FCNTL	SYSCALL_DEFINE3(fcntl, ...) SYSCALL_DEFINE3(fcntl64, ...)	struct file *file unsigned int cmd unsigned long arg
ID_FILE_SET_FOWNER	__f_setown()	struct file *file
ID_FILE_SEND_SIGIOTASK	sigio_perm()	struct task_struct *tsk struct fown_struct *fown int sig
ID_FILE_RECEIVE	scm_detach_fds_compat() scm_detach_fds()	struct file *file
ID_FILE_OPEN	do_dentry_open()	struct file *file const struct cred *cred

Ganchos em operações sobre processos

Gancho	Descrição	Parâmetros
ID_TASK_CREATE	Intercepa a função `copy_process()`. Gancho posicionado imediatamente antes da criação de um novo processo	unsigned long clone_flags

Gancho	Descrição	Parâmetros
ID_TASK_FREE	Não trata-se de gancho restritivo (return void). Este gancho é posicionado imediatamente antes da função free_task(), responsável pela liberação de recursos relacionados a processos	struct task_struct *task
ID_CRED_ALLOC_BLANK	Gancho para credenciais LSM (cred->security).	struct cred *cred
	Tratador deve alocar memória para o campo cred->security de tal forma que o gancho cred_transfer() não retorne o erro -ENOMEM	gfp_t gfp
ID_CRED_FREE	Gancho para credenciais LSM (cred->security). Tratador deve desalocar (cred->security)	struct cred *cred
ID_CRED_PREPARE	Gancho para credenciais LSM (cred->security).	struct cred *new
	Tratador deve definir new como cópia de old	struct cred *old
		gfp_t gfp
ID_CRED_TRANSFER	Gancho para credenciais LSM (cred->security).	struct cred *new
	Tratador deve transferir credenciais de old para new	struct cred *old
ID_KERNEL_ACT_AS	Intercepta a função set_security_override(). Não trata-se de um gancho restritivo. Permite ao módulo LSM ajustar as	struct cred *new
	credenciais de um processo com um novo rótudo, definindo um novo contexto de execução	u32 secid
ID_KERNEL_CREATE_FILES_AS	Intercepta a função set_create_files_as(). Não trata-se de um gancho restritivo. Permite ao módulo LSM a criação de	struct cred *new
	arquivos com mesmo contexto de segurança (UID e GID), do objeto inode	struct inode *inode
ID_KERNEL_MODULE_REQUEST	Gancho permissivo. Permite ao *kernel* utilizar a função __request_module(), responsável por tentar carregar um LKM utilizando o carregador de módulos disponível em espaço de usuário	char *kmod_name
ID_TASK_FIX_SETUID	Não trata-se de um gancho retristivo. Permite ao módulo LSM atualizar atributos de identidade do processo	struct cred *new
	current sempre que uma chamada de sistema da família set*uid for executada. As alterações devem informações	const struct cred *old
	devem ser feitas em cred *new, e não em current->cred	int flags
ID_TASK_SETPGID	Intercepta a chamada de sistema getpgid()	struct task_struct *p
		pit_t pgid

Gancho	**Descrição**	**Parâmetros**
`ID_TASK_GETPGID`	Intercepta a chamada de sistema `getpgid()`	`struct task_struct *p`
`ID_TASK_GETSID`	Intercepta a chamada de sistema `getsid()`	`struct task_struct *p`
`ID_TASK_GETSECID`	Não trata-se de um gancho restritivo (`return void`). Permite a diferentes subsistemas do Linux recuperarem o rótulo `secid` definido pelo módulo LSM corrente para processo `p`.	`struct task_struct *p` `u32 secid`
`TASK_SETNICE`	Intercepta a chamada de sistema `nice()`, responsável por mudança de prioridade de escalonamento de processos	`struct task_struct *p` `int nice`
`ID_TASK_SETIOPRIO`	Intercepta a função `set_task_ioprio()`, checagens de permissões da chamada de sistema `ioprio_set()`	`struct task_struct *p` `int ioprio`
`ID_TASK_GETIOPRIO`	Intercepta a função `get_task_ioprio()`, checagens de pemissões da chamada de sistema `ioprio_get()`	`struct task_struct *p`

`ID_TASK_SETRLIMIT`	Intercepta a função `do_prlimit()`, utilizada pelas chamadas de sistema `prlimit64()` e `setrlimit()`	`struct task_struct *p`
		`unsigned int resource`
		`struct rlimit *new_rlim`

`ID_TASK_SETSCHEDULER`	Intercepta as chamadas de sistema `sched_setscheduler`, `sched_setaffinity` e função `cpuset_can_attach` do subsistema `cpuset` do *framework* TCG	`struct task_struct *p`
`ID_TASK_GETSCHEDULER`	Intercecpta as chamadas de sistema `sched_rr_get_interval()`, `sched_getscheduler()`, `sched_getaffinity()` e `sched_getparam()`	`struct task_struct *p`
`ID_TASK_MOVEMEMORY`	Intercepta as chamadas de sistema `move_pages()` e `migrate_pages`, responsáveis pela migração de páginas de memória do processo entre diferentes nós de memória física.	`struct task_struct *p`

`ID_TASK_KILL`	Intercepta a função `check_kill_permission()`, responsável por checagens de permissões para execução de chamadas de sistema de envio de sinais para entre processos, tais como `kill()`, `tkill()` e `tgkill()`	`struct task_struct *p`
		`struct siging *info`
		`int sig`
		`u32 secid`

`ID_TASK_WAIT`	Intercepta a função `wait_consider_task()`, responsável pela autorização do envio de informações de mudança de estado (chamada de sistema `wait()`) do processo `p` para o seu pai (`p->parent`)	`struct task_struct *p`

Gancho	Descrição	Parâmetros
ID_TASK_PRCTL	Intercepta a chamada de sistema prctl(), reponsável pela execução de operações sobre processos	int option
		unsigned long arg2
		unsigned long arg3
		unsigned long arg4
		unsigned long arg5
ID_TASK_TO_INODE	Intercepta a função proc_pid_make_inode(), responsável pela criação de arquivos virtuais com informações de processos em /proc/<PID>/	task struct *p
		struct inode *inode

Ganchos em operações sobre mensgens Netlink

Gancho	Função interceptada	Parâmetros
ID_NETLINK_SEND	netlink_sendmsg()	struct sock *sk
		struct sk_buff *skb

Ganchos em operações sobre *Unix domain networking*

Gancho	Função interceptada	Parâmetros
ID_UNIX_STREAM_CONNECT	unix_stream_connect()	struct sock *sock
		struct sock *other
		struct sock *newsk
ID_UNIX_MAY_SEND	unix_dgram_connect() unix_dgram_sendmsg()	struct socket *sock
		struct socket *other

Ganchos em operações sobre *sockets*

Gancho	Função interceptada	Parâmetros
ID_SOCKET_CREATE	sock_create_lite() __sock_create()	int family
		int type
		int protocol
		int kern
ID_SOCKET_POST_CREATE	sock_create_lite() __sock_create()	struct socket *sock
		int family
		int type
		int protocol
		int kern

Gancho	**Função interceptada**	**Parâmetros**
ID_SOCKET_BIND	SYSCALL_DEFINE3(bind, ...)	struct socket *sock
		struct sockaddr *address
		int addrlen
ID_SOCKET_CONNECT	SYSCALL_DEFINE3(connect, ...)	struct socket *sock
		struct sockaddr *address
		int addrlen
ID_SOCKET_LISTEN	SYSCALL_DEFINE2(listen, ...)	struct socket *sock
		int backlog
ID_SOCKET_ACCEPT	SYSCALL_DEFINE4(accept4, ...)	struct socket *sock
		struct socket *newsock
ID_SOCKET_SENDMSG	__sock_sendmsg()	struct socket *sock
		struct msghdr *msg
		int size
ID_SOCKET_RECVMSG	__sock_recvmsg()	struct socket *sock
		struct msghdr *msg
		int size
		int flags
ID_SOCKET_GETSOCKNAME	SYSCALL_DEFINE3(getsockname, ...)	struct socket *sock
ID_SOCKET_GETPEERNAME	SYSCALL_DEFINE3(getpeername, ...)	struct socket *sock
ID_SOCKET_GETSOCKOPT	SYSCALL_DEFINE5(getsockopt, ...)	struct socket *sock
		int level
		int optname
ID_SOCKET_SETSOCKOPT	SYSCALL_DEFINE5(setsockopt, ...)	struct socket *sock
		int level
		int optname
ID_SOCKET_SHUTDOWN	SYSCALL_DEFINE2(shutdown, ...)	struct socket *sock
		int how
ID_SOCKET_SOCK_RCV_SKB	sk_filter()	struct sock *sk
		struct sk_buff *skb
ID_SOCKET_GETPEERSEC_STREAM	sock_getsockopt()	struct socket *sock
		char __user *optval
		int __user *optlen
		unsigned len
ID_SOCKET_GETPEERSEC_DGRAM	ip_cmsg_recv_security()	struct socket *sock
		struct sk_buff *skb
		u32 *secid
ID_SK_ALLOC_SECURITY	sk_prot_alloc()	struct sock *sk
		int family
		gfp_t priority
ID_SK_FREE_SECURITY	sk_prot_alloc() sk_prot_free()	struct sock *sk
ID_SK_CLONE_SECURITY	l2cap_sock_init() rfcomm_sock_init() sco_sock_init() sock_copy()	const struct sock *sk struct sock *newsk

Gancho	Função interceptada	Parâmetros
ID_SOCK_GRAFT	sock_graft()	struct sock *sk
		struct socket *parent
ID_INET_CONN_REQUEST	dccp_v4_conn_request()	struct sock *sk
	dccp_v6_conn_request()	struct sk_buff *skb
	cookie_v4_check()	struct rqs_sock *req
	tcp_v4_conn_request()	
	cookie_v6_check()	
	tcp_v6_conn_request()	
ID_INET_CSK_CLONE	inet_csk_clone_lock()	struct sock *newsk
		const struct rqst_sock *req
ID_INET_CONN_ESTABLISHED	tcp_finish_connect()	struct sock *sk
		struct sk_buff *skb
ID_SECMARK_RELABEL_PACKET	checkentry_lsm()	u32 secid
ID_SECURITY_SECMARK_REFCOUNT_INC	checkentry_lsm()	void
ID_SECURITY_SECMARK_REFCOUNT_DEC	secmark_tg_destroy()	void
ID_REQ_CLASSIFY_FLOW	dccp_v6_send_response()	const struct rqst_sock *req
	inet_csk_route_req()	struct flowi *fl
	inet_csk_route_child_sock()	
	cookie_v4_check()	
	inet6_csk_route_req()	
	cookie_v6_check()	
ID_TUN_DEV_CREATE	tun_set_iff()	void
ID_TUN_DEV_POST_CREATE	tun_set_iff()	struct sock *sk
ID_TUN_DEV_ATTACH	tun_set_iff()	struct sock *sk

Ganchos em operações XFRM

Gancho	Função interceptada	Parâmetros
ID_XFRM_POLICY_ALLOC_SECURITY	pfkey_spdadd() pfkey_spddelete() pfkey_compile_policy() copy_from_user_sec_ctx xfrm_get_policy xfrm_add_pol_expire	struct xfrm_sec_ctx **ctxp struct xfrm_user_sec_ctx *ctx
ID_XFRM_POLICY_CLONE_SECURITY	clone_policy()	struct xfrm_sec_ctx *old_ctx struct xfrm_sec_ctx **new_ctx
ID_XFRM_POLICY_FREE_SECURITY	pfkey_spddelete() xfrm_policy_destroy() xfrm_add_policy() xfrm_get_policy() xfrm_add_pol_expire()	struct xfrm_sec_ctx *ctx
ID_XFRM_POLICY_DELETE_SECURITY	xfrm_policy_bysel_ctx() xfrm_policy_byid() xfrm_policy_flush_secctx_check()	struct xfrm_sec_ctx *ctx
ID_XFRM_STATE_ALLOC_SECURITY	pfkey_msg2xfrm_state() xfrm_state_construct()	struct xfrm_state *x struct xfrm_user_sec_ctx *ctx u32 secid
ID_XFRM_STATE_FREE_SECURITY	xfrm_state_gc_destroy()	struct xfrm_state *x
ID_XFRM_STATE_DELETE_SECURITY	pfkey_delete() xfrm_state_flush_secctx_check xfrm_del_sa	struct xfrm_state *x
ID_XFRM_POLICY_LOOKUP	xfrm_policy_match() xfrm_sk_policy_lookup()	struct xfrm_sec_ctx *ctx u32 fl_secid u8 dir
ID_XFRM_STATE_POL_FLOW_MATCH	xfrm_state_look_at()	struct xfrm_state *x struct xfrm_policy *xp const struct flowi *fl
ID_XFRM_DECODE_SESSION	__xfrm_decode_session()	struct sk_buff *skb u32 *secid int ckall

Ganchos em operações de IPC

Gancho	Função interceptada	Parâmetros
ID_IPC_PERMISSION	ipcperms()	struct kern_ipc_perm *ipcp short flag
ID_IPC_GETSECID	__audit_ipc_obj()	struct kern_ipc_perm *ipcp u32 *secid

Ganchos em operações do sistema de gerenciamento de chaves do Linux

Gancho	Função interceptada	Parâmetros
ID_KEY_ALLOC	key_alloc()	struct key *key
		const struct cred *cred
		unsigned long flags
ID_KEY_FREE	key_gc_unused_keys()	struct key *key
ID_KEY_PERMISSION	key_task_permission()	key_ref_t key_ref
		const struct cred *cred
		key_perm_t perm
ID_KEY_GETSECURITY	keyctl_get_security()	struct key *key
		char **_buffer

Ganchos em operações de filas de mensagems de IPC

Gancho	Função interceptada	Parâmetros
ID_MSG_MSG_ALLOC_SECURITY	load_msg()	struct msg_msg *msg
ID_MSG_MSG_FREE_SECURITY	free_msg()	struct msg_msg *msg
ID_MSG_QUEUE_ALLOC_SECURITY	newque()	struct msg_queue *msq
ID_MSG_QUEUE_FREE_SECURITY	newque() freeque()	struct msg_queue *msq
ID_MSG_QUEUE_ASSOCIATE	msg_security()	struct msg_queue *msq
		int msqflg
ID_MSG_QUEUE_MSGCTL	SYSCALL_DEFINE3(msgctl, ...)	struct msg_queue *msq
		int cmd
ID_MSG_QUEUE_MSGSND	do_msgsnd()	struct msg_queue *msq
		struct msg_msg *msg
		int msqflg
ID_MSG_QUEUE_MSGRCV	pipelined_send() do_msgrcv()	struct msg_queue *msq
		struct msg_msg *msg
		struct task_struct *target
		long type
		int mode

Ganchos em operações de memória compartilhada

Gancho	Função interceptada	Parâmetros
ID_SHM_ALLOC_SECURITY	newseg()	struct shmid_kernel *shp
ID_SHM_FREE_SECURITY	shm_destroy() newseg()	struct shmid_kernel *shp

Gancho	Função interceptada	Parâmetros
ID_SHM_ASSOCIATE	shm_security()	struct shmid_kernel *shp
		int shmflg
ID_SHM_SHMCTL	SYSCALL_DEFINE3(shmctl, ...)	struct shmid_kernel *shp
		int cmd
ID_SHM_SHMAT	do_shmat()	struct shmid_kernel *shp
		char __user *shmaddr
		int shmflg

Ganchos em operações de Semáforos

Gancho	Função interceptada	Parâmetros
ID_SEM_ALLOC_SECURITY	newary()	struct sem_array *sma
ID_SEM_FREE_SECURITY	newary() freeary()	struct sem_array *sma
ID_SEM_ASSOCIATE	sem_security()	struct sem_array *sma
		int semflg
ID_SEM_SEMCTL	semctl_nolock() semctl_main() semctl_down()	struct sem_array *sma int cmd
ID_SEM_SEMOP	SYSCALL_DEFINE4(semtimedop, ...)	struct sem_array *sma
		struct sembuf *sops
		unsigned nsops
		int alter
ID_PTRACE_ACCESS_CHECK	__ptrace_may_access()	struct task_struct *child
		unsigned int mode
ID_PTRACE_TRACEME	ptrace_traceme()	struct task_struct *parent
ID_CAPGET		struct task_struct *target
		kernel_cap_t *effective
		kernel_cap_t *inheritable
		kernel_cap_t *permitted
ID_CAPSET	cap_get_target_pid()	struct cred *new
		struct cred *old
		kernel_cap_t *effective
		kernel_cap_t *inheritable
		kernel_cap_t *permitted
ID_CAPABLE	pci_read_config() has_ns_capability() ns_capable()	const struct cred *cred struct user_namespace *ns int cap

Gancho	Função interceptada	Parâmetros
ID_SYSLOG	devkmsg_open() do_syslog	int type
ID_SETTIME	SYSCALL_DEFINE1(stime, ...)	const struct timespec *ts
		const struct timezone *tz
ID_VM_ENOUGH_MEMORY	SYSCALL_DEFINE1(swapoff, ...)	struct mm_struct *mm long pages

Ganchos em operações de auditoria

Gancho	Função interceptada	Parâmetros
ID_AUDIT_RULE_INIT		u32 field
		u32 opchar *rulestr
		void **lsmrule
ID_AUDIT_RULE_KNOWN		struct audit_krule *krule
ID_AUDIT_RULE_MATCH		u32 secid
		u32 field
		u32 op
		void *lsmrule
		struct audit_context *actx
ID_AUDIT_RULE_FREE		void *lsmrule
ID_INODE_NOTIFYSECCTX		struct inode *inode
		void *ctx
		u32 ctxlen
ID_INODE_SETSECCTX		struct dentry *dentry
		void *ctx
		u32 ctxlen
ID_INODE_GETSECCTX		struct inode *inode
		void **ctx
		u32 *ctxlen