MARCOS AURELIO PCHEK LAUREANO

UMA ABORDAGEM PARA A PROTEÇÃO DE DETECTORES DE INTRUSÃO BASEADA EM MÁQUINAS VIRTUAIS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática Aplicada.

CURITIBA

2004

MARCOS AURELIO PCHEK LAUREANO

UMA ABORDAGEM PARA A PROTEÇÃO DE DETECTORES DE INTRUSÃO BASEADA EM MÁQUINAS VIRTUAIS

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática Aplicada.

Área de Concentração: *Metodologia e Técnicas de Computação*

Orientador: Prof. Dr. Carlos Maziero

CURITIBA

2004

Laureano, Marcos Aurelio Pchek
Uma Abordagem para a Proteção de Detectores de Intrusão Baseada em Máquinas Virtuais. Curitiba, 2004. 103p.

Dissertação de Mestrado – Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática Aplicada.

1. Segurança 2. Máquinas Virtuais 3. Detecção de Intrusão 4. Sistemas Operacionais. I. Pontifícia Universidade Católica do Paraná. Centro de Ciências Exatas e de Tecnologia. Programa de Pós-Graduação em Informática Aplicada

Dedico este trabalho à minha família.

Agradecimentos

Agradeço a minha esposa Margarete, que sempre teve paciência e compreensão nos momentos que fiquei ausente e pelo apoio que ela me deu durante todo o tempo. Ao meu filho Luiz Otávio, que sempre pedia colo e carinho nos momentos mais complicados deste trabalho (se não fosse por ele, eu teria ficado louco).

Agradeço ao José Nauiack, que acreditou em mim (sem ao menos me conhecer direito) e conseguiu uma liberação inédita na empresa onde trabalho e aos demais colegas do HSBC que me auxiliaram no decorrer deste trabalho.

Agradeço ao Carlos Maziero, pelas oportunidades que ele me proporcionou durante todo o desenvolvimento deste trabalho.

Sumário

AGRADECIMENTOS ..... **IV**

SUMÁRIO ..... **V**

LISTA DE FIGURAS ..... **VIII**

LISTA DE TABELAS ..... **IX**

LISTA DE ABREVIAÇÕES ..... **X**

RESUMO ..... **XI**

ABSTRACT ..... **XII**

CAPÍTULO 1 ..... **1**

INTRODUÇÃO 1
1.1 Motivação 1
1.2 Proposta 2
1.3 Organização do Trabalho 2

CAPÍTULO 2 ..... **4**

MÁQUINAS VIRTUAIS 4
2.1 Definição 4
2.2 Tipos de Máquinas Virtuais 5
2.2.1 Máquinas Virtuais de Tipo I 6
2.2.2 Máquinas Virtuais do Tipo II 6
2.2.3 Abordagens Híbridas 7
2.3 Propriedades de Monitores de Máquinas Virtuais 9
2.4 Uso de Máquinas Virtuais 9
2.4.1 Benefícios 9
2.4.2 Dificuldades 10
2.5 Exemplos de Máquinas Virtuais 12
2.5.1 User-Mode Linux 12
2.5.2 VMware 13
2.5.3 Denali 15
2.5.4 Xen 16
2.5.5 Java Virtual Machine 17
2.6 Conclusão 19

CAPÍTULO 3 ..... **20**

DETECÇÃO DE INTRUSÃO 20
3.1 Classificação de Detectores de Intrusão 20
3.1.2 Quanto à Origem dos Dados 21

3.2 Limitações 22
3.3 Detecção por Assinatura 23
3.4 Detecção por Anomalia 24
3.5 Ferramentas Existentes 25
3.5.1 - EMERALD 25
3.5.2 - NetSTAT 26
3.5.3 - BRO 27
3.5.4 - Outros exemplos 28
3.5.5 - Produtos GOTS (Government Off-the-Shelf) – Produtos não disponíveis comercialmente 29
3.6 Conclusão 30

CAPÍTULO 4 32

CHAMADAS DE SISTEMA E SEGURANÇA 32
4.1 Conceituação 33
4.2 Ataques a Chamadas de Sistema 35
4.2.1 Denial of Service 36
4.2.2 Buffer Overflow 37
4.2.3 Rootkits 37
4.3 Detecção de Intrusão por Análise de Chamadas de Sistema 38
4.4 Classificação de Chamadas de Sistema 39
4.5 Conclusão 42

CAPÍTULO 5 43

PROTEÇÃO DE DETECTORES DE INTRUSÃO USANDO MÁQUINAS VIRTUAIS 43
5.1 Problema 43
5.2 Proposta 43
5.3 Funcionamento da Proposta 44
5.3.1 O Processo de Aprendizado 46
5.3.2 Monitoração e Resposta 49
5.3.3 Controle de Acesso 51
5.3.4 Capacidade de reação 52
5.4 Benefícios e Limitações da Proposta 52
5.5 Trabalhos Correlatos 54
5.5.1 Projeto Revirt 54
5.5.2 Projeto VMI IDS 54
5.5.3 LIDS – Linux Intrusion Detection System 55
5.6 Conclusão 56

CAPÍTULO 6 58

IMPLEMENTAÇÃO E RESULTADOS 58
6.1 Alterações na Máquina Virtual 58
6.2 Avaliação do Protótipo 60
6.2.1 Custo 61
6.2.2 Rootkits Utilizados Para os Testes 62
6.2.3 Testes por Anomalia na Seqüência de Chamadas de Sistema 63
6.2.4 Testes com Utilização de ACLs 64
6.3 Análise dos Resultados Obtidos 65
6.4 Considerações Finais 66

6.5 Conclusão 66
CAPÍTULO 7 67
CONCLUSÃO 67
REFERÊNCIAS BIBLIOGRÁFICAS 69
APÊNDICE A 78
EXEMPLO DE SEQÜÊNCIA DE CHAMADAS DE SISTEMA 78
APÊNDICE B 85
SEQÜÊNCIA DE CHAMADAS DE SISTEMA DO COMANDO LOGIN 85

Lista de Figuras

Figura 2.1 – Máquina Virtual de Tipo I **6**

Figura 2.2 – Máquina Virtual do Tipo II **7**

Figura 2.3 – Abordagem Híbrida para Tipo I **7**

Figura 2.4 – Abordagem Híbrida para Tipo II **8**

Figura 2.5 – Virtualização da chamada de sistema [DIK00] **13**

Figura: 2.6 – Ambiente Java Típico **18**

Figura 4.1 – 11 passos para fazer uma chamada read(arq, buffer, nbytes) [TAN03] **34**

Figura 5.1 – Modelo Genérico **44**

Figura 5.2 – Funcionamento do IDS **46**

Figura 5.3 – IDS em modo Aprendizado **47**

Figura 5.4 – IDS em modo Monitoração **50**

Figura 6.1 – Alterações no Kernel do Monitor **59**

Figura 6.2 – Protótipo Implementado **60**

Lista de Tabelas

Tabela 4.1 – Chamadas de Sistema comuns [SIL00a, SIL00b] 35

Tabela 4.2 – Exemplo de fork bomb 36

Tabela 4.3 – Categoria de Chamadas de Sistema 40

Tabela 4.4 – Categoria de Chamadas de Sistema Classificadas por nível de ameaça 41

Tabela 5.1 – Seqüência de chamadas de sistema sem parâmetros do comando who 47

Tabela 5.2 – Conjunto de chamadas de sistema agrupadas 48

Tabela 5.3 – Relação usuário versus processo 49

Tabela 5.4 – Chamadas de sistema negadas aos processos suspeitos 51

Tabela 5.5 – Exemplo de ACL 52

Tabela 6.1 – Tempo Médio de Execução (em milisegundos) 61

Tabela 6.2 – Acréscimo de Tempo 62

Tabela 6.3 – Rootkits utilizados para validar o VMIDS 62

Tabela 6.4 – Seqüências de chamadas de sistema inválidas 64

Tabela 6.5 – ACL gerada pelo VMIDS durante a fase de aprendizado 64

Tabela 6.6 – Lista de usuários autorizados 65

Tabela 6.7 – Lista de processos autorizados 65

Tabela B.1 – Seqüências de chamadas de sistema do comando login original. 85

Tabela B.2 – Seqüências de chamadas de sistema do comando login alterado 88

Lista de Abreviações

ACD	*Access Control Database*
ACL	*Access Control List*
BIOS	*Basic Input-Output System*
CMDS	*Computer Misuse Detection System*
CPU	*Central Processing Unit*
DNS	*Domain Name Service*
DoS	*Denial of Service*
EMERALD	*Event Monitoring Enabling Responses to Anomalous Live Disturbances*
E/S	Entrada e Saída
GPL	*General Public Licence*
GOTS	*Government Off-the-Shelf*
HIDS	*Host Intrusion Detection System*
IDS	*Intrusion Detection System*
JVM	*Java Virtual Machine*
LIDS	*Linux Intrusion Detection System*
NFR	*Network Flight Recorder*
NIDS	*Network Intrusion Detection System*
PDA	*Personal Digital Assistant*
PID	*Process Identification*
RPC	*Remote Procedure Call*
SSL	*Secure Socket Layer*
UID	*User Id*
UML	*User-Mode Linux*
USB	*Universal Serial Bus*
VM	*Virtual Machine ou Máquina Virtual*
VMIIDS	*Virtual Machine Introspection Intrusion Detection System*
VMIDS	*Virtual Machine Intrusion Detection System*
VMM	*Virtual Machine Monitor ou Monitor de Máquinas Virtuais*

Resumo

Diversas ferramentas contribuem para aumentar a segurança de um sistema computacional. Dentre elas, destacam-se os sistemas de detecção de intrusão. Tais sistemas monitoram continuamente a atividade em uma rede ou servidor, buscando evidências de intrusão. Entretanto, detectores de intrusão baseados em *host* são particularmente vulneráveis, pois devem ser instalados nas próprias máquinas a monitorar e podem ser desativados ou modificados por invasores bem sucedidos. Este trabalho propõe e implementa uma arquitetura para a aplicação confiável e robusta de detectores de intrusão baseados em *host*, através da utilização do conceito de máquina virtual. A utilização de máquinas virtuais vem se tornando uma alternativa interessante para vários sistemas de computação, por suas vantagens em custos e portabilidade. Como demonstrado neste trabalho, o conceito de máquina virtual também pode ser empregado para melhorar a segurança de um sistema computacional contras ataques a seus serviços. A proposta aqui apresentada faz uso da separação de espaços de execução provida por um ambiente de máquinas virtuais para separar o sistema de detecção de intrusão do sistema a monitorar. Com isso, o detector de intrusão se torna invisível e inacessível a eventuais invasores. A implementação da arquitetura proposta e os testes realizados demonstraram a viabilidade dessa solução.

Palavras-Chave: Segurança, Máquina Virtual, Detecção de Intrusão, Sistemas Operacionais.

Abstract

Several tools contribute to improve the security of a computing system. Among them, intrusion detection systems stand out. Such systems continuously watch the activity on a network or server, looking for intrusion evidences. However, host-based intrusion detectors are particularly vulnerable, as they can be disabled or tampered by successful intruders. This work proposes and implements an architecture for the robust and reliable use of host-based intrusion detectors, through the application of the virtual machine concept. Virtual machine environments are becoming an interesting alternative for several computing systems, because of their advantages in terms of cost and portability. As shown in this work, the virtual machine concept can also be used to improve the security of a computing system against attacks to its services. The architecture proposal presented here makes use of the execution spaces separation provided by a virtual machine environment, in order to separate the intrusion detection system from the system under monitoring. In consequence, the intrusion detector becomes invisible and unaccessible to intruders. The architecture implementation and the tests performed show the viability of this solution.

Keywords: Security, Virtual Machine, Intrusion Detection System, Operating System.

Capítulo 1

Introdução

1.1 Motivação

A Internet mudou as formas como se usam sistemas de informação. As possibilidades e oportunidades de utilização são muito mais amplas que em sistemas fechados, assim como os riscos à privacidade e integridade da informação. Portanto, é muito importante que mecanismos de segurança de sistemas de informação sejam projetados de maneira a prevenir acessos não autorizados aos recursos e dados destes sistemas. Mesmo com a tecnologia disponível no momento, é praticamente impossível impedir que isso aconteça. Não existe um mecanismo único que forneça uma solução para esse problema.

Uma ferramenta muito importante neste cenário é conhecida como Sistema de Detecção de Intrusão (IDS – *Intrusion Detection System*). A detecção de intrusão é um método para detectar ataques, uso malicioso ou inadequado de recursos em um sistema computacional. A detecção pode ser feita por meio da monitoração do ambiente de execução interno a um *host* (IDS baseado em *host*), pela monitoração do tráfego de rede (IDS baseado em rede) ou por uma combinação dos dois anteriores (sistemas híbridos).

Essas ferramentas continuamente coletam e analisam dados dos sistemas e/ou redes monitoradas, buscando detectar tentativas de intrusão. Caso um ataque seja detectado, são gerados alertas que permitam a correção de eventuais problemas; também podem ser ativadas contra-medidas para atuar sobre o tráfego de rede de forma a interromper o ataque.

Detectores de intrusão baseados em *host* são particularmente vulneráveis a ataques, pois devem ser instalados nas próprias máquinas a monitorar, a fim de poder coletar os dados de análise. Portanto, podem ser facilmente desativados ou modificados por invasores bem

sucedidos, visando ocultar sua presença.

1.2 Proposta

A utilização de máquinas virtuais vem se tornando uma alternativa interessante para vários sistemas de computação, por suas vantagens em custos e portabilidade, principalmente em consolidação de servidores. Em um ambiente de máquinas virtuais, um software monitor executa sobre um sistema operacional anfitrião (ou diretamente sobre o *hardware*) e suporta a execução de vários sistemas operacionais convidados isolados entre si. O conceito de máquina virtual pode ser utilizado também para melhorar a segurança de um sistema computacional contra ataques a seus serviços.

A proposta deste trabalho é definir e implementar uma arquitetura confiável para a aplicação de sistemas de detecção de intrusão baseados em *host*. Isso é obtido através da execução dos processos de aplicação a monitorar em máquinas virtuais (ou seja, dentro de um sistema operacional convidado) e a implantação dos sistemas de detecção e resposta a intrusões externo à máquina virtual (ou seja, no sistema operacional anfitrião). Esta separação protege o sistema de detecção de intrusão, uma vez que o mesmo ficará inacessível aos processos do sistema operacional convidado (e a possíveis intrusos).

Através de alterações na máquina virtual, é possível coletar informações de forma transparente aos processos e seus usuários. Esses dados são então enviados a processos externos para a detecção de intrusões. Utilizando uma base histórica (criada a partir da observação do sistema virtual) para comparação, o sistema de detecção de intrusão procura por desvios de comportamento nos processos ou nos usuários do sistema. Através da interceptação das chamadas de sistema realizadas pelos processos do sistema convidado, o sistema de detecção de intrusão pode comandar ações para impedir o acesso a recursos do sistema convidado caso seja detectada alguma anomalia no comportamento do sistema.

1.3 Organização do Trabalho

Este trabalho é composto de 7 capítulos, incluindo este. O capítulo 2 apresenta os principais conceitos relacionados a máquinas virtuais e a descrição de alguns ambientes de máquinas virtuais em uso atualmente; o capítulo 3 conceitua e detalha aspectos de detecção de intrusão; o capítulo 4 conceitua chamadas de sistema e analisa alguns aspectos de segurança relacionados às mesmas; o capítulo 5 contém a proposta da dissertação; o capítulo 6 apresenta

o protótipo implementado, os ensaios realizados e a análise crítica dos resultados obtidos; finalmente, o capítulo 7 conclui a dissertação, apontando os benefícios alcançados e os trabalhos futuros relacionados à proposta e sua implementação.

Capítulo 2

Máquinas Virtuais

O conceito de máquina virtual não é novo – suas origens remetem ao início da história dos computadores, no final dos anos 50 e início dos anos 60 [GOL73; VAR89]. As máquinas virtuais foram originalmente desenvolvidas para centralizar os sistemas de computador utilizados no ambiente VM/370 da IBM [GOL74, GOL79]. Naquele sistema, cada máquina virtual simula uma réplica física da máquina real e os usuários têm a ilusão que o sistema está disponível para seu uso exclusivo [SUG01].

A utilização de máquinas virtuais está se tornando uma alternativa para vários sistemas de computação, pelas vantagens em custos e portabilidade [BLU02; SIL00a], inclusive em sistemas de segurança [HON03].

2.1 Definição

Uma máquina virtual (*Virtual Machine* - VM) é definida em [POP74] como "uma duplicata eficiente e isolada de uma máquina real". A IBM define uma VM como uma cópia totalmente protegida e isolada de um sistema físico [GOL74, GOL79, SUG01].

Uma máquina real é formada por vários componentes físicos que fornecem operações para o sistema operacional e suas aplicações. Iniciando pelo núcleo do sistema real, o processador central (CPU) e o *chipset* da placa-mãe fornecem um conjunto de instruções e outros elementos fundamentais para o processamento de dados, alocação de memória e processamento de Entrada e Saída (E/S). Ao fundo estão os dispositivos e os recursos tais como a memória, o vídeo, o áudio, os discos rígidos, os CDROMs, e as portas (USB, paralela, serial). Em uma máquina real, a BIOS ou *devices drivers* específicos fornecem as operações

de baixo nível para que um sistema operacional possa acessar os vários recursos da placa-mãe, memória ou serviços de E/S.

Um **emulador** é o oposto da máquina real. O emulador implementa todas as instruções realizadas pela máquina real em um ambiente abstrato de software, possibilitando executar um aplicativo de uma plataforma em outra, por exemplo, um aplicativo do Windows executando no Linux ou um aplicativo *i386* executando em uma plataforma *Sparc*. Infelizmente, um emulador perde muito em eficiência ao traduzir cada instrução da máquina real. Além disso, emuladores são bastante complexos, pois geralmente necessitam simular a quase totalidade das instruções do processador e demais características do *hardware* que os circundam [MAL73].

A funcionalidade e o nível de abstração de uma VM encontra-se em uma posição intermediária entre uma máquina real e um emulador, na forma em que os recursos de *hardware* e de controle são abstraídos e usados pelas aplicações. Uma VM é um ambiente criado por um monitor de máquina virtual (*Virtual Machine Monitor* – VMM), também denominado "sistema operacional para sistemas operacionais" [KEL91]. O monitor pode criar uma ou mais VMs sobre uma única máquina real. Enquanto um emulador fornece uma camada de abstração completa entre o sistema em execução e o *hardware*, um monitor fornece uma interface (através da multiplexação do *hardware*) que é idêntica ao *hardware* subjacente e controla uma ou mais VMs. Cada VM, que recebe uma cópia (virtual) do computador, fornece facilidades para uma aplicação ou um "sistema convidado" que acredita estar executando sobre um ambiente convencional com acesso direto ao *hardware*. Um emulador também fornece uma abstração do *hardware* idêntico ao que está em uso, mas também pode simular outros diferentes do atual [KIN02].

2.2 Tipos de Máquinas Virtuais

Existem basicamente duas abordagens para a construção de sistemas de máquinas virtuais: o tipo I, onde o monitor é implementado entre o *hardware* e os **sistemas convidados** (*guest system*), e o tipo II, onde o monitor é implementado como um processo de um sistema operacional real subjacente, denominado **sistema anfitrião** (*host system*) [SUG01, KIN02].

As figuras 2.1 e 2.2 ilustram a organização tradicional de um sistema de máquinas virtuais. Para maximizar o desempenho, o monitor sempre que possível, permite que a VM execute diretamente sobre o *hardware*, em modo usuário. O monitor retoma o controle

sempre que a VM tenta executar uma operação que possa afetar o correto funcionamento do sistema, o conjunto de operações de outras VMs ou do próprio *hardware*. O monitor simula com segurança a operação antes de retornar o controle à VM.

2.2.1 Máquinas Virtuais de Tipo I

O monitor tem o controle do *hardware* e cria um ambiente de máquinas virtuais, cada VM se comporta como uma máquina física completa que pode executar o seu próprio sistema operacional, semelhante a um sistema operacional tradicional que está no controle da máquina. O resultado da completa virtualização da máquina é um conjunto de computadores virtuais executando sobre o mesmo sistema físico.

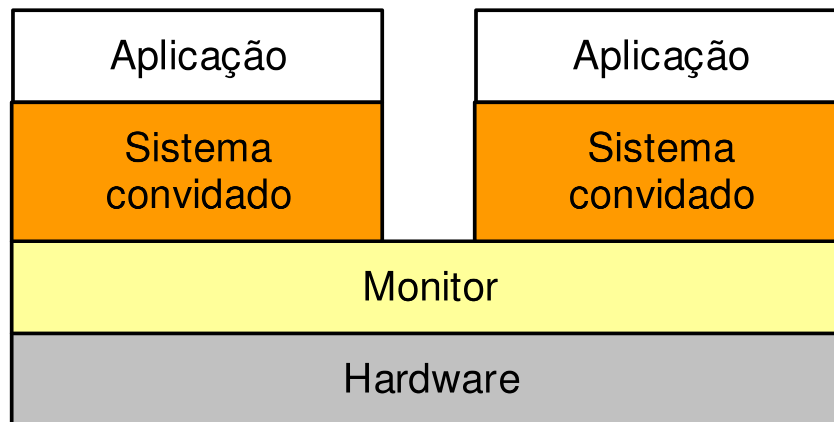

Figura 2.1 – Máquina Virtual de Tipo I

2.2.2 Máquinas Virtuais do Tipo II

O monitor executa sobre um sistema anfitrião, como um processo num sistema real. O monitor de Tipo II funciona de forma análoga ao de Tipo I, sendo a sua maior diferença a existência de um sistema abaixo deste. Neste modelo, o monitor simula todas as operações que o sistema anfitrião controlaria.

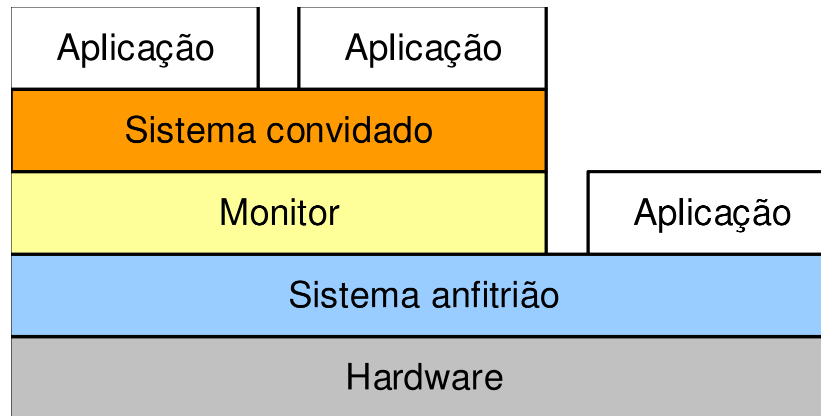

Figura 2.2 – Máquina Virtual do Tipo II

2.2.3 Abordagens Híbridas

Os monitores de tipo I e II raramente são usados em sua forma conceitual em implementações reais. Na prática, várias otimizações são inseridas nas arquiteturas apresentadas, com o objetivo principal de melhorar o desempenho das aplicações nos sistemas convidados. Como os pontos cruciais do desempenho dos sistemas de máquinas virtuais são as operações de E/S, as principais otimizações utilizadas em sistemas de produção dizem respeito a essas operações. Quatro otimizações são usuais:

Em monitores de tipo I:

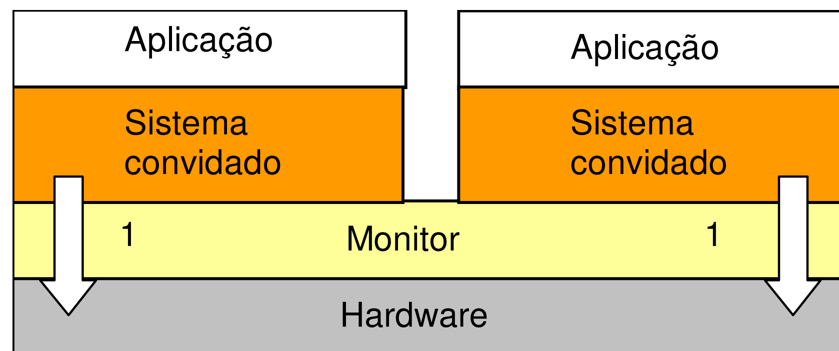

Figura 2.3 – Abordagem Híbrida para Tipo I

1 – O sistema convidado (*guest system*) acessa diretamente o *hardware*. Essa forma de acesso é implementada através de modificações no núcleo do sistema convidado e no monitor. Essa otimização é implementada, por exemplo, no subsistema de gerência de memória do ambiente Xen [BAR03a, BAR03b].

Em monitores de tipo II:

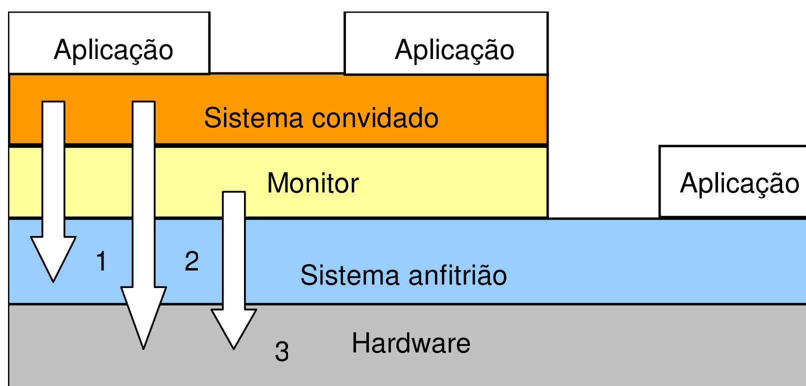

Figura 2.4 – Abordagem Híbrida para Tipo II

1 – O sistema convidado (*guest system*) acessa diretamente o sistema anfitrião (*host system*). Essa otimização é implementada pelo monitor, oferecendo partes da API do sistema anfitrião ao sistema convidado. Um exemplo dessa otimização é a implementação do sistema de arquivos no VMWare [VM99]: ao invés de reconstruir integralmente o sistema de arquivos sobre um dispositivo virtual provido pelo monitor, o sistema convidado faz uso da implementação de sistema de arquivos nativa existente no sistema anfitrião.

2 – O sistema convidado (*guest system*) acessa diretamente o hardware. Essa otimização é implementada parcialmente pelo monitor e parcialmente pelo sistema anfitrião, através de um *device driver* específico. Um exemplo típico dessa otimização é o acesso direto a dispositivos físicos como leitor de CDs, *hardware* gráfico e interface de rede provida pelo sistema VMWare [VM99] aos sistemas operacionais convidados.

3 – O monitor acessa diretamente o *hardware*. Neste caso, um *device driver* específico é instalado no sistema anfitrião, oferecendo ao monitor uma interface de baixo nível para acesso ao *hardware* subjacente. Essa abordagem é usada pelos sistemas VMware [VM99] e UML [KIN02, KIN03].

Essas otimizações levam a arquiteturas que diferem dos modelos básicos I e II, sendo por isso chamadas de *abordagens híbridas*.

2.3 Propriedades de Monitores de Máquinas Virtuais

Os monitores possuem algumas propriedades [ATT73, BEL73, POP74, SIR99] que podem ser utilizadas na segurança de sistemas [GAR03]:

- **Isolamento** – Um *software* em execução em uma VM não acessa ou modifica outro *software* em execução no monitor ou em outra VM;
- **Inspeção** – O monitor tem acesso e controle sobre todas as informações do estado da VM, como estado da CPU, conteúdo de memória, eventos, etc;
- **Interposição** – O monitor pode intercalar ou acrescentar instruções em certas operações de uma VM, como por exemplo, quando da execução de instruções privilegiadas por parte da VM;
- **Eficiência** – Instruções inofensivas podem ser executadas diretamente no *hardware*, pois não irão afetar outras VMs ou aplicações;
- **Gerenciabilidade** – Como cada VM é uma entidade independente das demais, a administração das diversas instâncias é simplificada e centralizada.

Além destas propriedades, um monitor oferece outras, como o encapsulamento de estado, que pode ser utilizado para construir *checkpoints* do estado da VM. Estados salvos têm vários usos, como *rollback* e análise *post-mortem*.

Hardwares virtualizáveis [GOL73, GOL74, POP74, GOL79], como as máquinas *mainframe* da IBM [GOL74, GOL79], têm uma propriedade, chamada *execução direta*, que permite que a estes sistemas obtenham, com a utilização de VMs, o desempenho similar ao de um sistema convencional equivalente.

2.4 Uso de Máquinas Virtuais

Ao longo dos anos, as máquinas virtuais vem sendo utilizadas com vários fins, como processamento distribuído e segurança. Um uso freqüente de sistemas baseados em máquinas virtuais é a chamada “consolidação de servidores”: em vez da utilização de vários equipamentos com seus respectivos sistemas operacionais, utiliza-se somente um computador, com máquinas virtuais abrigando os vários sistemas operacionais e suas respectivas aplicações e serviços [SUG01,FRA03].

2.4.1 Benefícios

Muitos dos benefícios das máquinas virtuais utilizadas no ambiente dos *mainframes*,

também podem ser obtidos nos computadores pessoais [SUG01]. A abordagem da IBM, que define uma VM como uma cópia totalmente protegida e isolada de um sistema físico, permite que testes de sistemas na fase de desenvolvimento, não prejudiquem os demais usuários em caso de um travamento do equipamento virtualizado [GOL74, GOL79, SUG01]. Nos *mainframes*, as máquinas virtuais também são utilizadas para *timesharing* ou divisão de recursos através das diversas aplicações.

Em [GOL73; OZD94; AGR99; SIL00a; SUG01, BLU02, FRA03] são relacionados algumas vantagens para a utilização de máquinas virtuais em sistemas de computação:

- Facilitar o aperfeiçoamento e testes de novos sistemas operacionais;
- Auxiliar no ensino prático de sistemas operacionais e programação ao permitir a execução de vários sistemas para comparação no mesmo equipamento;
- Executar diferentes sistemas operacionais sobre o mesmo *hardware*, simultaneamente;
- Simular configurações e situações diferentes do mundo real, como por exemplo, mais memória disponível ou a presença de outros dispositivos de E/S;
- Simular alterações e falhas no *hardware* para testes ou re-configuração de um sistema operacional, provendo confiabilidade e escalabilidade para as aplicações;
- Garantir a portabilidade das aplicações legadas (que executariam sobre uma VM simulando o sistema operacional original);
- Desenvolvimento de novas aplicações para diversas plataformas, garantindo a portabilidade destas aplicações;
- Diminuição de custos com *hardware*, através da consolidação de servidores;
- Facilidades no gerenciamento, migração e replicação de computadores, aplicações ou sistemas operacionais;
- Prover um serviço dedicado para um cliente específico com segurança e confiabilidade.

2.4.2 Dificuldades

Segundo [SUG01], além do custo do processo de virtualização em si, existem outras dificuldades para a ampla utilização de VMs em ambientes de produção:

- **Processador Não Virtualizado** – A arquitetura dos processadores Intel 32 bits não permite naturalmente a virtualização [INT98]. Em [POP74] foi demonstrado que uma arquitetura pode suportar VMs somente se todas as instruções que podem inspecionar

ou modificar o estado privilegiado da máquina forem executados em modo mais privilegiado e puderem ser interceptados. O processador Intel de 32 bits não se encontra nesta situação, pois não é possível virtualizar o processador para executar todas as operações em um modo menos privilegiado;

- **Diversidade de Equipamentos** – Existe uma grande quantidade de equipamentos disponíveis (características da arquitetura aberta do PC). Em uma execução tradicional, o monitor teria que controlar todos estes dispositivos, o que requer um esforço de programação grande por parte dos desenvolvedores de monitores;
- **Pré-existência de *softwares*** – Ao contrário de *mainframes* que são configurados e controlados por administradores de sistema, os *desktops* e *workstations* normalmente já vêm com um sistema operacional instalado e pré-configurado, e que normalmente é ajustado pelo usuário final. Neste ambiente, é extremamente importante permitir que um usuário possa utilizar a tecnologia das VMs, mas sem perder a facilidade de continuar utilizando seu sistema operacional padrão e aplicações.

A principal desvantagem do uso de máquinas virtuais é o custo adicional de execução dos processos na máquina virtual em comparação com a máquina real. Esse custo é muito variável, podendo chegar a 50% ou mais em plataformas sem suporte de *hardware* a virtualização, como os PCs de plataforma Intel [INT98, VM99, DIK00, BLU02]. Esse problema inexiste em ambientes de *hardware* com suporte a virtualização, como é o caso de *mainframes* [GOL74, GOL79]. Todavia, pesquisas recentes têm obtido a redução desse custo a patamares abaixo de 20%, graças, sobretudo, a ajustes no código do sistema anfitrião [WHI02; KIN02; KIN03]. Outra técnica utilizada é a reescrita "*on-the-fly*" de partes do código executável das aplicações, inserindo pontos de interceptação do controle antes/após as instruções privilegiadas cuja virtualização não é permitida na plataforma Intel de 32 bits [VM99]. Um exemplo desse avanço é o projeto Xen [BAR03a, BAR03b], no qual foram obtidos custos da ordem de 3% para a virtualização de ambientes Linux, FreeBSD e Windows XP. Esse trabalho abre muitas perspectivas na utilização de máquinas virtuais em ambientes de produção.

O trabalho [SUG01] descreve uma proposta para a ampla utilização de máquinas virtuais em computadores pessoais.

2.5 Exemplos de Máquinas Virtuais

Nesta seção será apresentada a arquitetura das máquinas virtuais que mais se destacam atualmente, seja pela sua relevância comercial ou acadêmica. Além dos exemplos aqui apresentados, diversos outros sistemas e projetos de pesquisa relacionados ao tema podem ser encontrados na literatura.

2.5.1 User-Mode Linux

O *User-Mode Linux* foi proposto em [DIK00] como uma alternativa de uso de máquinas virtuais no ambiente Linux. O *kernel* do Linux foi portado de forma a poder executar sobre si mesmo, como um conjunto de processos do próprio Linux. O resultado é um *user space* separado e isolado na forma de uma VM que utiliza a simulação de *hardware* construída a partir dos serviços providos pelo sistema anfitrião. Essa VM é capaz de executar todos os serviços e aplicações disponíveis para o sistema anfitrião.

O *User-Mode Linux* é uma VM de Tipo II, ou seja, executa na forma de um processo no sistema anfitrião, e os processos em execução na VM não têm acesso aos recursos do sistema anfitrião diretamente. O monitor é um processo único que controla a execução de todas as VMs. Há um processo no anfitrião para cada instância da máquina virtual, e um único processo no anfitrião para o monitor.

A maior dificuldade na implementação do *User-Mode Linux* foi encontrar maneiras para virtualizar todas as capacidades do *hardware* para as chamadas de sistema do Linux, sobretudo a distinção entre o modo privilegiado do *kernel* e o modo não-privilegiado de usuário. Um código somente pode estar em modo privilegiado se é confiável o suficiente para permitir pleno acesso ao *hardware*, como o próprio *kernel* do sistema operacional. O *User-Mode Linux* deve possuir uma distinção de privilégios equivalente para permitir que o seu *kernel* tenha acesso às chamadas de sistema do sistema anfitrião quando os seus próprios processos solicitarem este acesso, ao mesmo tempo em que impede os mesmos de acessar diretamente os recursos subjacentes. Esta distinção de privilégio foi implementada com o mecanismo de interceptação de chamadas do próprio Linux fornecida pela chamada *ptrace*[1].

Usando *ptrace*, o monitor ganha o controle de todas as chamadas de sistema de entrada e saída geradas nas VMs. Todos os sinais gerados ou enviados às VMs também são interceptados. A chamada *ptrace* também é utilizada para manipular o contexto do convidado.

---

[1] O *ptrace* é uma chamada de sistema que permite observar e controlar a execução de outros processos.

O *User-Mode Linux* utiliza o sistema anfitrião para operações de E/S. Como a VM é um processo no sistema anfitrião, a troca entre duas instâncias de VMs é rápida, assim como a troca entre dois processos do anfitrião. Entretanto, modificações nos *devices drivers* do sistema convidado foram necessárias para a otimização da troca de contexto.

A virtualização das chamadas de sistema é implementada através de uma *thread* de rastreamento (Figura 2.5) que intercepta e redireciona todas as chamadas de sistema para o *kernel* virtual. Os passos decorrentes do rastreamento são representados pelas linhas cheias, as linhas tracejadas ilustram a execução de uma chamada de sistema sem o rastreamento (sistema Linux tradicional). Este identifica a chamada de sistema e os seus argumentos, anula a chamada e modifica estas informações no anfitrião. Para executar a chamada de sistema no *user space* do usuário, o processo troca de contexto e executa a chamada na pilha do *kernel*.

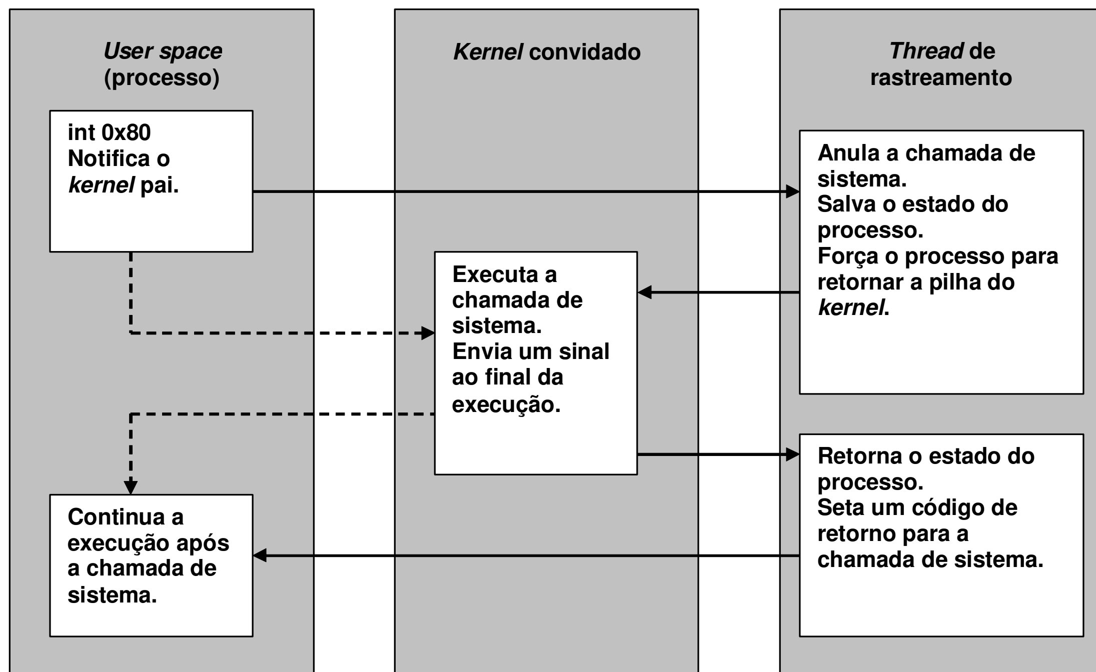

Figura 2.5 – Virtualização da chamada de sistema [DIK00]

O *User-Mode Linux* já está disponível na versão 2.6 do *kernel* do Linux, ou seja, ele entrou na árvore de desenvolvimento do *kernel*, portanto melhorias na sua arquitetura deverão surgir no futuro, ampliando sua utilização e aceitação para diversas aplicações.

2.5.2 VMware

O VMware [VM99] é hoje a máquina virtual para a plataforma *x86* de uso mais

difundido. Provendo uma implementação completa da interface *x86* ao sistema convidado, o VMware é uma ferramenta útil em diversas aplicações. Embora essa interface seja extremamente genérica para o sistema convidado, acaba conduzindo a um monitor mais complexo. Como podem existir vários sistemas operacionais em execução no mesmo *hardware*, o monitor tem que emular certas instruções para representar corretamente um processador virtual em cada máquina virtual; as instruções que devem ser emuladas são chamadas de *instruções sensíveis*. Por razões de desempenho, as máquinas virtuais geralmente confiam no mecanismo de *trap* (armadilha) do processador para executar instruções sensíveis. Porém, os processadores *x86* não capturam todas as instruções sensíveis e um trabalho adicional deve ser realizado. Para controlar as instruções sensíveis que não foram capturadas, o VMware utiliza uma técnica chamada re-escrita binária (*binary rewriting*). Com esta técnica, todas as instruções são examinadas antes de serem executadas, e o monitor insere pontos de parada no lugar das instruções sensíveis. Quando executado, o ponto de parada faz com que o processador capture a instrução do monitor. Essa técnica acrescenta complexidade ao monitor do VMware, mas provê um conjunto completo de instruções *x86* para a interface do sistema convidado.

Por razões de desempenho, o monitor do VMware utiliza uma abordagem híbrida para implementar a interface do monitor com as VMs. O controle de exceção e gerenciamento de memória é realizado através da manipulação direta do *hardware*, mas para simplificar o monitor, o controle de E/S é do sistema anfitrião. Através do uso de abstrações para suportar a E/S, o monitor evita manter *device drivers*, algo que os sistemas operacionais já implementam adequadamente. Esta simplificação causou uma perda de desempenho em versões mais antigas do VMware, mas foram adotadas otimizações para diminuir seus efeitos e melhorar o desempenho de E/S.

A gerência de memória no VMware é feita diretamente pelo sistema convidado. Para garantir que não ocorra nenhuma colisão de memória entre o sistema convidado e o real, o VMware aloca uma parte da memória para uso exclusivo, então o sistema convidado utiliza essa memória previamente alocada.

Para controlar o sistema convidado, o VMware implementa serviços de interrupção para todas as interrupções do sistema convidado. Sempre que uma exceção é causada no convidado, é examinado primeiro pelo monitor. As interrupções de E/S são remetidas para o sistema anfitrião, para que sejam controladas corretamente. As exceções geradas pelas

aplicações no sistema convidado (como as chamadas de sistema, por exemplo) são remetidas para o sistema convidado.

2.5.3 Denali

O projeto Denali [WSD02, KIN02] implementa um ambiente de máquinas virtuais cujo principal objetivo é reduzir o custo de virtualização da máquina virtual, provendo um forte isolamento entre diferentes instâncias de máquinas virtuais em execução, para simplificar o monitor. Estes objetivos são alcançados através de uma técnica chamada para-virtualização (*para-virtualization*). A técnica envolve a modificação da interface do monitor convidado onde algumas instruções são adicionadas e outras retiradas da VM. Com a utilização de certas instruções *x86*, não é necessário o uso da técnica de re-escrita binária. Como a plataforma *x86* é complexa e requer uma grande quantidade de emulações para garantir a compatibilidade, a remoção de algumas instruções acaba por simplificar o código do monitor.

Como parte da arquitetura virtual do Denali, o controle de E/S virtual é mantido pelo monitor. Por exemplo, um pacote inteiro *Ethernet* pode ser enviado ao monitor, pelo sistema convidado, através do uso de uma única instrução. Isto reduz significativamente o número de chamadas que monitor recebe e simplifica a arquitetura do sistema convidado.

A VM Denali executa diretamente no *hardware* sem necessidade de um sistema operacional anfitrião (ou seja, usa uma arquitetura de tipo I). Conseqüentemente, o monitor do Denali tem que prover *devices drivers* para todo o *hardware* da plataforma adotada. Pela implementação dos *devices drivers* no monitor, o Denali força políticas de multiplexação total do *hardware*. Isto garante o isolamento entre diferentes instâncias de máquinas virtuais, mas dificulta a implementação do monitor.

Não existe o conceito de memória virtual no ambiente Denali. O sistema convidado é executado em um único espaço de memória privado. Esta abordagem simplifica o monitor, mas a falta de proteção de memória dentro da VM limita a capacidade do sistema convidado. Esta situação impõe alterações complexas no projeto de como as aplicações devem ser construídas.

A manipulação de interrupções também é diferenciada. Em vez de tratar interrupções quando elas acontecem, elas são colocadas numa fila até que a VM as execute. Isto reduz o número de interrupções que o monitor necessita tratar, e conseqüentemente, o custo de virtualização torna-se menor.

Para suportar um alto desempenho e simplificar a arquitetura do sistema convidado e o código do monitor, o ambiente Denali sacrifica a generecidade do código do monitor, a ponto de perder a compatibilidade com a plataforma *x86*, até mesmo com aplicações que executam sob o modo de usuário. A diminuição do custo na virtualização é muito interessante e faz com que o ambiente Denali seja uma referência para pesquisas futuras nessa área, mas a perda de compatibilidade faz com que muitas aplicações existentes não possam ser utilizadas.

2.5.4 Xen

O ambiente Xen é um monitor de Tipo I para a plataforma *x86*. Suporta múltiplos sistemas convidados simultaneamente com bom desempenho e isolamento. Sua arquitetura está descrita em [BAR03a, BAR03b], sendo o componente principal de um projeto mais amplo chamado *XenoServers* [FRA03] que consiste em construir uma infra-estrutura para computação distribuída.

A proposta do ambiente Xen é suportar aplicações sem a necessidade de alterações, múltiplos sistemas operacionais convidados e a cooperação entre estes sistemas, mas com o máximo de desempenho possível. Utilizando a técnica de *para-virtualization*, proposta anteriormente pelo projeto Denali [WSD02, KIN02], o monitor foi alterado em três grandes aspectos no sistema:

Gerenciamento de Memória:

- **Segmentação** – Não é possível instalar descritores de segmentação com privilégio total e o descritor não pode sobrepor a extremidade superior do endereço de memória linear;
- **Paginação** – O sistema convidado tem direito de leitura as tabelas de páginas de memória, mas toda modificação deve ser validada pelo monitor.

Gerência CPU:

- **Proteção** – O sistema convidado deve executar em um nível de privilégio menor que o do monitor;
- **Exceções** – O sistema convidado registra uma tabela de *handlers* para controle das exceções no monitor. Excetuando-se o controle de exceção de páginas (controlado pelo monitor), os demais permanecem sem alterações;
- **Chamadas de Sistema (*System Calls*)** – O sistema convidado pode instalar um controlador “rápido” para as chamadas de sistema, permitindo chamadas diretas

de uma aplicação para o *kernel* do sistema convidado e evitando a interferência do monitor;

- **Interrupções** – As interrupções de *hardware* são substituídas por um controle de eventos mais leve, similar ao do projeto Denali;
- **Controle de Tempo** – O sistema convidado tem acesso a uma interface que fornece o tempo "real" e o "virtual".

Gerência de dispositivos de E/S:

- **Rede, Disco, etc.** – Os dados são transferidos de forma elegante e simples, utilizando E/S assíncrono. É implementando um mecanismo de eventos para substituir as notificações de *hardware*.

Os sistemas convidados precisam ser alterados para executar sob o Xen. Conforme demonstrado no trabalho [BAR03a, BAR03b], o custo e impacto das alterações nos sistemas convidados são baixos e a diminuição do custo da virtualização compensa essas alterações.

O monitor Xen se encontra em um acentuado grau de maturidade e pode ser utilizado em sistemas de produção; o seu código fonte está liberado sob a licença *GNU General Public Licence* (GPL). Atualmente, o ambiente Xen suporta os sistemas Windows XP, Linux e Unix (baseado no NetBSD).

2.5.5 Java Virtual Machine

É comum a implementação de linguagens de programação utilizando uma máquina virtual, um bom exemplo de utilização de máquina virtual em programação é a máquina *P-Code* do *UCSD Pascal* [LIN99]. Neste caso, a linguagem de programação não é traduzida diretamente para uma linguagem compreensível ao processador, mas sim em um pseudocódigo. Para a execução, outro programa é utilizado para *interpretar* esse código. A estrutura da máquina *P-Code* é orientada a *stacks*, ou seja, a maioria das instruções examina a pilha para colocar e retirar suas operações e valores.

O trabalho [LIN99] descreve a máquina virtual Java (*Java Virtual Machine – JVM*). A JVM é à base da plataforma Java e Java 2. A JVM é um componente tecnológico responsável pela independência do *hardware* e sistemas operacionais, tem código compilado de tamanho reduzido e com a habilidade de proteger o usuário de códigos maliciosos. A JVM é baseada na estrutura de *stacks*.

A JVM é uma abstração (emulador) dos sistemas de computação. Semelhante ao computador real, é dotado de um conjunto de instruções que manipula várias áreas de

memória ou instruções de *hardware* em tempo de execução.

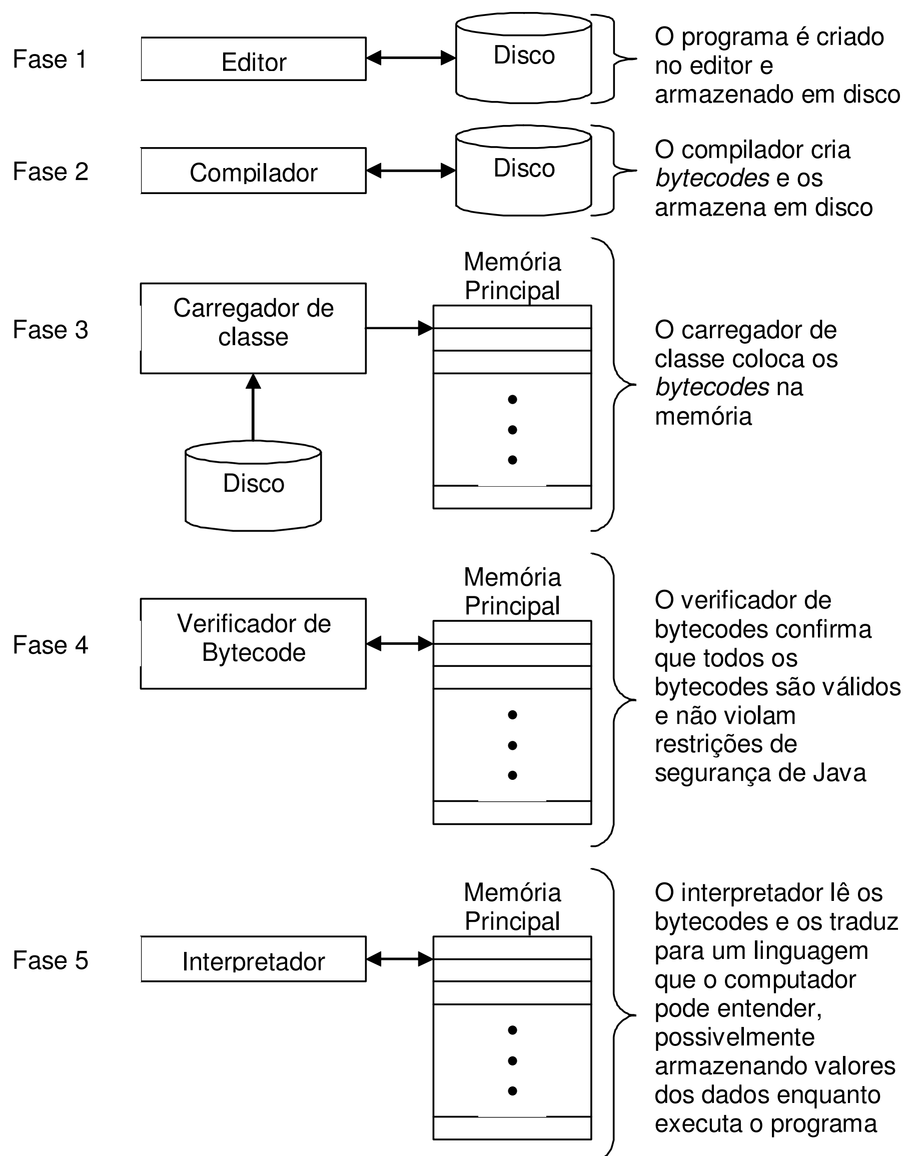

Figura: 2.6 – Ambiente Java Típico

O primeiro protótipo da implementação da JVM, através da Sun Microsystems Inc., emulava um conjunto de instruções através de uma JVM instalada em um dispositivo *handheld* semelhante a um *Personal Digital Assistant* (PDA). A implementação atual da JVM (padronizada pela Sun) emula aplicações Java em sistemas operacionais como Windows e Solaris, entretanto a JVM não implementa tecnologias ligadas a um único *hardware* ou sistema operacional. A JVM tem a capacidade de executar aplicações diretamente sobre um

sistema operacional ou através de micro-códigos implementados em *hardware* (*smartcard*, por exemplo).

A JVM não conhece a linguagem Java, somente um formato binário em particular que é o arquivo no formato *class*. Este arquivo contém um conjunto de instruções (ou *bytecodes*) e tabelas de símbolos para a JVM. A arquitetura dos *bytecodes* é neutra (independente de *hardware* ou sistema operacional), existindo uma JVM portada para cada sistema operacional ou *hardware* anfitrião. A JVM lê os *bytecodes*, verifica a sua integridade e executa os códigos envolvidos. Este processo pode ser visto na Figura 2.6.

A JVM deve ser considerada basicamente um emulador, na medida em que cria um ambiente de execução abstrato para aplicações Java e não um ambiente de máquinas virtuais similar aos demais descritos neste capítulo.

2.6 Conclusão

Como visto nas seções anteriores, a utilização de máquinas virtuais está se tornando uma alternativa para vários sistemas de computação, pelas vantagens em custos e portabilidade. Os desafios para a utilização de máquinas virtuais vêm sendo superados devidos às várias pesquisas sendo realizadas. O conceito de máquina virtual também pode ser empregado para melhorar a segurança de um sistema computacional contras ataques a seus serviços, como sugerido em [CHE01].

Capítulo 3

Detecção de Intrusão

A maneira mais comum para descobrir intrusões é a utilização dos dados das auditorias gerados pelos sistemas operacionais e ordenados em ordem cronológica de acontecimento, sendo possível à inspeção manual destes registros, o que não é uma prática viável, pois estes arquivos de *logs* apresentam tamanhos consideráveis.

Nos últimos anos, a tecnologia de detecção de intrusão (*Intrusion Detection System* – IDS) tem se mostrado uma grande aliada dos administradores de segurança. Basicamente, o que tais sistemas fazem é tentar reconhecer um comportamento ou uma ação intrusiva, através da análise das informações disponíveis em um sistema de computação ou rede, para alertar um administrador e/ou automaticamente disparar contra-medidas. Para realizar a detecção, várias tecnologias estão sendo empregadas em produtos comerciais ou em projetos de pesquisas, as tecnologias utilizadas incluem análise estatística, inferência, inteligência artificial, *data mining*, redes neurais e diversas outras [ALL99].

Um IDS automatiza a tarefa de analisar dados da auditoria. Estes dados são extremamente úteis, pois podem ser usados para estabelecer a culpabilidade do atacante e na maioria das vezes é o único modo de descobrir uma atividade sem autorização, detectar a extensão dos danos e prevenir tal ataque no futuro, tornando desta forma o IDS uma ferramenta extremamente valiosa para análises em tempo real e também após a ocorrência de um ataque.

3.1 Classificação de Detectores de Intrusão

O IDS tem como principal objetivo detectar se alguém está tentando entrar em um

sistema ou se algum usuário legítimo está fazendo mau uso do mesmo. Esta ferramenta é executada constantemente em *background* e somente gera uma notificação quando detecta alguma ocorrência que seja suspeita ou ilegal. Os sistemas em uso podem ser classificados com relação a sua forma de monitoração (origem dos dados) e aos mecanismos (algoritmos) de detecção utilizados.

3.1.2 Quanto à Origem dos Dados

Existem basicamente dois tipos de implementação de ferramentas IDS:

- ***Host Based IDS*** **(HIDS)** – são instalados em servidores para alertar e identificar ataques e tentativas de acesso indevido à própria máquina, sendo mais empregados nos casos em que a segurança está focada em informações contidas em um servidor;
- ***Network Based IDS*** **(NIDS)** – são instalados em máquinas responsáveis por identificar ataques direcionados a toda a rede, monitorando o conteúdo dos pacotes de rede e seus detalhes como informações de cabeçalhos e protocolos.

Os sistemas NIDS podem monitorar diversos computadores simultaneamente. Todavia, sua eficácia diminui na medida em que o tamanho e a velocidade da rede aumenta, pela necessidade de analisar os pacotes mais rapidamente. Além disso, o uso de protocolos cifrados (baseados em SSL – *Secure Socket Layer*) torna o conteúdo dos pacotes opaco ao IDS. A velocidade da rede e o uso de criptografia não são problemas para os sistemas HIDS. Todavia, como esse sistema é instalado na própria máquina a monitorar, pode ser desativado por um invasor bem-sucedido. Existem IDS que trabalham de forma híbrida, ou seja, combinando as duas técnicas citadas anteriormente [ALL99].

3.1.3 Quanto à Forma de Detecção

Muitas ferramentas de IDS realizam suas operações a partir da análise de padrões do sistema operacional e da rede tais como: utilização de CPU, E/S de disco, uso de memória, atividades dos usuários, número de tentativas de *login*, número de conexões, volume de dados trafegando no segmento de rede entre outros. Estes dados formam uma base de informação sobre a utilização do sistema em vários momentos ao longo do dia. Algumas ferramentas possuem bases com padrões de ataque (assinaturas) previamente constituído, permitindo também a configuração das informações já existentes bem como inclusão de novos parâmetros. As técnicas usadas para detectar intrusões podem ser classificadas em:

- **Detecção por assinatura** – os dados coletados são comparados com uma base de registros de ataques conhecidos (assinaturas). Por exemplo, o sistema pode vasculhar os pacotes de rede procurando seqüências de *bytes* que caracterizem um ataque de *buffer overflow* contra o servidor WWW Apache;
- **Detecção por anomalia** – os dados coletados são comparados com registros históricos da atividade considerada normal do sistema. Desvios da normalidade são sinalizados como ameaças. Os modelos estatísticos mais utilizados em detectores de intrusão por anomalia foram propostos em [DEN87];
- **Detecção Híbrida** – o mecanismo de análise combina as duas abordagens anteriores, buscando detectar ataques conhecidos e comportamentos anormais.

A detecção por assinatura é a técnica mais empregada nos sistemas de produção atuais. Um exemplo de IDS baseado em assinatura é o SNORT [ROE99, KOZ03]. Os sistemas antivírus também adotam a detecção por assinatura. A detecção de intrusão por anomalia ainda é pouco usada em sistemas de produção.

3.2 Limitações

Com as informações extraídas dos sistemas de computação, uma ferramenta de IDS pode identificar as tentativas de intrusão e até mesmo registrar a técnica utilizada. Um IDS não é perfeito, podendo ocorrer:

- **Falsos positivos** – ocorrem quando a ferramenta classifica uma ação como uma possível intrusão, quando na verdade trata-se de uma ação legítima; Um bom exemplo de falso positivo ocorre quando um servidor *web* recebe vários pacotes do tipo `SYN` e o IDS conclui que se trata de um ataque do tipo `SYN FLOOD`;
- **Falsos negativos** – ocorrem quando uma intrusão real acontece, mas a ferramenta não detecta, pois considera a ação legítima;
- **Subversão** – ocorre quando o intruso (através de ataques) modifica a operação da ferramenta de IDS para forçar a ocorrência de falso negativo. Após a invasão a uma máquina ser bem sucedida, um atacante pode enviar informações falsas como se fossem ações legítimas da máquina invadida. Outro exemplo: Um invasor pode, fornecer uma quantidade (superior a capacidade de análise) de dados para que o IDS investigue, tornando-o inoperante permanentemente ou por um certo período de tempo, o que bastaria para uma invasão sem alarmes.

Gerar falsos positivos tende ao comprometimento da confiança na ferramenta. Por outro lado, a quantidade de falsos negativos coloca dúvidas sobre a eficácia do IDS.

3.3 Detecção por Assinatura

A detecção por Assinatura analisa a atividade do sistema, procurando por eventos ou conjunto de eventos que corresponda a um determinado padrão que caracterize um ataque conhecido. Em geral, essa abordagem gera um número menor de falsos positivos se comparados à detecção por Anomalia. Por se basear em padrões a detecção é mais rápida e específica, possibilitando ações diferenciadas para cada caso, mas em compensação é ineficiente frente a novos padrões de ataques.

Uma assinatura tradicional contém uma seqüência de *bytes* que representam ou especificam um ataque. Se essa assinatura estiver em um pacote de rede capturado, é uma indicação de um provável ataque. Os sistemas NIDS utilizam esta abordam para a detecção de intrusão, através da utilização de expressões regulares, análise de contexto ou linguagens de assinatura, os pacotes de rede são analisados e comparado com uma base de dados de assinaturas [SOM03]. O SNORT [ROE99] é um dos NIDS mais utilizados atualmente [KOZ03]. O SNORT mantém regras de detecção de intrusão em uma lista através do controle de uma cadeia de cabeçalhos (*headers*) e opções (*options*). Através da análise dos cabeçalhos dos pacotes de rede, chega-se às possíveis opções de análise. A cadeia de regras é pesquisada recursivamente para cada pacote de rede que chega ao computador. A maior vantagem do SNORT é a simplicidade para se escrever novas regras de detecção, possibilitando a tomada de ações ou simplesmente a notificação do administrador de rede [ROE99].

Atualmente, a melhor tecnologia que descreve a detecção por assinatura são os programas de antivírus [NAC97]. Segundo [SHI00], a definição para vírus e seus equivalentes:

- **Vírus** – Um programa escondido, que se auto-replica, um código geralmente com lógica maliciosa que se propaga através da infecção – introduz uma cópia de si mesmo em outros computadores;
- ***Trojan Horse*** – Um programa de computador que parece ter uma função útil, mas tem também uma função às vezes escondida e potencialmente maliciosa que burla mecanismos da segurança, explorando autorizações legítimas de uma entidade do sistema que invoca o programa;

- ***Worm*** – Um programa de computador que funciona de forma independente, pode propagar uma versão completa de si mesma para outros anfitriões de uma rede, e pode consumir recursos do computador de forma destrutiva.

Os programas antivírus possuem uma base das assinaturas dos vírus existentes e todo novo arquivo ou e-mail passa por um processo de *scanner* para verificar sua integridade. Se uma assinatura for detectada (provável infecção), o arquivo ou e-mail pode ser recusado, desinfectado ou colocado sob "quarentena".

3.4 Detecção por Anomalia

A detecção de anomalia caracteriza como ataque os padrões de comportamento considerados incomuns em um sistema ou rede. Para caracterizar um comportamento como anormal, são construídos perfis com base em dados coletados durante um período de operação normal. O IDS então utiliza um conjunto de métricas para determinar quando a informação monitorada difere dos perfis estabelecidos. A detecção de anomalia, por ser baseada no comportamento normal do sistema e não nas características específicas do ataque, como na detecção por assinatura, é capaz de detectar ataques desconhecidos. Além disso, em alguns casos, as informações produzidas pela detecção de um ataque podem ser utilizadas como assinatura em futuras detecções. Entretanto, a maior desvantagem dessa abordagem é que geralmente a análise produz um grande número de falsos positivos. Isso acontece, porque o comportamento normal de um sistema pode variar de acordo com o tempo, através da adição de novos usuários e da utilização de novas aplicações.

Em um dos trabalhos de referência nesta área, Dorothy Denning define em [DEN87] quatro modelos estatísticos que podem ser utilizados em um detector de intrusão por anomalia. Cada modelo descrito na seqüência é considerado apropriado para um tipo particular de métrica.

- **Modelo operacional** – este modelo aplica-se a métricas como, por exemplo, contadores de eventos para o número de falhas de *login* em um determinado intervalo de tempo. O modelo compara a métrica a um limiar definido, identificando uma anomalia quando a métrica excede o valor limite. Esse modelo pode ser aplicado tanto na detecção por anomalia quanto na detecção por assinatura;
- **Modelo de média e desvio padrão** – este modelo propõe uma caracterização

clássica de média e desvio padrão para os dados. Uma nova observação de comportamento é identificada como anormal se ela encontra-se fora de um intervalo de confiança. Esse intervalo de confiança é definido como sendo *d* desvios-padrão da média, para algum parâmetro *d*. Denning sugere que essa caracterização seja aplicável a métricas do tipo contadores de eventos, intervalos de tempo e medidas de recursos;

- **Modelo multivalorado** – este modelo é uma extensão ao modelo de média e desvio padrão, baseia-se na correlação entre duas ou mais métricas. Desse modo, ao invés de basear a detecção de uma anomalia estritamente em uma métrica, essa detecção é baseada na correlação dessa métrica com alguma outra medida;
- **Modelo de processo de *Markov*** – este modelo é mais complexo e limitado a contadores de eventos. Segundo o mesmo, o detector considera cada tipo diferente de evento de auditoria como uma variável de estado e usa uma matriz de transição de estados para caracterizar as freqüências com que ocorrem as transições entre os estados. Uma nova observação de comportamento é definida como anormal se sua probabilidade, determinada pelo estado anterior e pelo valor na matriz de transição de estados, for muito baixa. Esse modelo permite que o detector identifique seqüências não usuais de comandos e eventos, introduzindo a noção de análise de fluxos de eventos com memória de estado.

3.5 Ferramentas Existentes

A tecnologia de IDS ainda é imatura e dinâmica (está continuamente em desenvolvimento e pesquisa). Nesta seção será apresentando os sistemas de detecção de intrusão que mais se destacam atualmente, seja pela sua relevância comercial ou acadêmica. Além dos exemplos aqui apresentados, diversos outros sistemas e projetos de pesquisa relacionados ao tema podem ser encontrados na literatura.

3.5.1 - EMERALD

O EMERALD – *Event Monitoring Enabling Responses to Anomalous Live Disturbances* (Monitoração de Eventos Ativando Respostas a Perturbações Anômalas *on-line*), desenvolvida pela SRI Internacional, verifica se houve uma intrusão baseando em desvios de comportamento do usuário (anomalias) e padrões de intrusão conhecidos

(assinaturas). A meta principal do projeto EMERALD é trabalhar com redes de empresas grandes (heterogêneas). Estes ambientes são difíceis de monitorar e de analisar devido à diversificação da informação que trafega pela rede. O EMERALD estrutura os usuários em um conjunto de domínios independentemente administrados. Cada conjunto provê uma cobertura de serviços de rede (`ftp`, `http`, `telnet`) que podem ter relações de confiança e políticas de segurança diferentes entre si [POR96, POR97].

A estrutura hierárquica provê três níveis de análise: monitores de serviço, domínio e empresa. Estes monitores possuem a mesma arquitetura básica: um conjunto de mecanismos de perfil (para descobertas de anomalia), mecanismos de assinatura e um componente determinador que integra os resultados gerados pelos mecanismos. É possível configurar e personalizar cada nível.

No nível mais baixo, o monitor de serviço suporta a detecção de intrusão para os componentes individuais e serviços de rede dentro de um domínio, sondando ou verificando *logs* e eventos, verificando assinaturas e realizando análises estatísticas. Os monitores de domínio integram a informação dos monitores de serviço para prover uma visão de invasões, enquanto os monitores de empresa executam uma análise interdomínio para avaliar as ameaças sob uma perspectiva global.

O NIDES [AND95] (que é mantido pela mesma empresa) demonstrou técnicas de análise estatísticas que poderiam ser efetivas com usuários ou aplicações. A monitoração de aplicações (`anonymous ftp`, por exemplo), era efetiva se menos perfis de aplicação fossem exigidos. O EMERALD generaliza a técnica de perfil pela abstração do que é um perfil, separando gerenciamento de perfil de análise de perfil. O EMERALD está em continuo desenvolvimento, sendo um exemplo de rumo que futuras ferramentas de IDS podem tomar.

3.5.2 - NetSTAT

O NetSTAT [VIG98] é a mais recente ferramenta de uma linha de ferramentas de investigação "STAT" produzida pela Universidade da Califórnia em Santa Bárbara. Este IDS explora o uso da análise de transição de estados para descobrir a intrusão em tempo real.

Sistemas HIDS analisam se houve uma invasão a partir da análise de trilhas de auditoria. Porém, na análise STAT [POR92] a informação de trilha de auditoria é transformada por um *analisador* que filtra e abstrai as informações que são recolhidas na trilha de auditoria. Estas abstrações, que são mais adequadas para análise, portabilidade e compreensão humana, são chamadas de assinaturas de estados. A análise da assinatura

modifica a seqüência de estados, cada mudança de estado deixa o sistema mais próximo de identificar uma intrusão. Seqüências de intrusão são definidas pelos diversos estados que são capturados neste sistema baseados em regras.

A aplicação inicial do método era um sistema HIDS desenvolvido para UNIX e chamado de USTAT. Era composto de um pré-processador, uma base de conhecimento (ações e regras), um mecanismo de inferência e um mecanismo de decisão [ILG93].

O NetSTAT está sob desenvolvimento e difere dos sistemas NIDS. O NetSTAT é composto de sondas que agem remotamente em cada sub-rede, caso alguma sonda identifique algum componente de intrusão, um evento é enviado as outras sondas interessadas para adquirir mais detalhes sobre a intrusão. Desta forma é possível identificar intrusões em sub-redes. As sondas são suportadas por um analisador, que é responsável pela administração da base de dados (base de conhecimento). É o analisador que determina qual e como os eventos serão monitorados.

3.5.3 - BRO

O BRO [PAX98] é uma ferramenta de investigação que foi desenvolvida pelo *Lawrence Livermore National Laboratory*. Foi construído, em parte, para explorar a robustez de ferramentas de IDS, isto é, avaliando quais características fazem um IDS resistir a ataques contra si mesmo. As metas do projeto abrangem:

- Monitoração *high-load* (capacidade de monitor altos tráfegos de rede);
- Notificação em tempo real;
- Separação de políticas de filtros, identificação e reação aos eventos. Facilita a aplicação e manutenção do sistema;
- Um amplo banco de dados com relação a ataques conhecidos e habilidade de acrescentar novos ataques a esta base;
- Habilidade para repelir ataques contra si mesmo.

O BRO trabalha com uma hierarquia de três níveis, na camada mais baixa é utilizada uma biblioteca chamada *libpcap*, que extrai pacotes da rede associados aos protocolos `finger`, `ftp`, `portmapper` e `telnet` que são os protocolos sobre os quais o BRO trabalha atualmente. A camada de evento executa verificações de integridade nos cabeçalhos (*headers*) dos pacotes, que verifica se deve ser feita uma análise mais profunda no pacote ou não. Na terceira camada os pacotes passam por um *script* que verifica as políticas de

segurança.

Embora, o BRO só monitore as aplicações citadas anteriormente, novas aplicações podem ser adicionadas a partir de uma derivação de uma classe em C++, somente devem ser acrescentadas algumas informações que correspondem à nova aplicação a ser monitorada.

3.5.4 - Outros exemplos

O trabalho [ALL99] cita outros exemplos de IDS disponíveis atualmente, alguns são sistemas comerciais outros são de domínio público. Exemplos comerciais:

- **CMDS** (*Computer Misuse Detection System*) – Sistema HIDS que verifica invasões através de mudança de perfil (anomalia) ou de assinatura;
- **NetProwler** – Sistema HIDS que verifica assinaturas. Permite que o usuário adicione novas assinaturas de ataque;
- **NetRanger** – É um sistema NIDS. Opera em tempo real e é escalável. Os sensores são espalhados pela rede e conversam com o software principal. Permite a análise de três categorias de ataques: ataques nomeados (ataques com um nome específico), gerais (variantes de ataques específicos) e extraordinários (com algo grau de complexidade);
- **Centrax** – Sistema HIDS que permite verificar pacotes criptografados. Pode reagir localmente a ameaças em tempo real e cada ataque pode ter um padrão de resposta diferente (como encerramento de uma conexão à máquina atacada);
- **RealSecure** – Outro exemplo de IDS em tempo real. Baseado numa arquitetura de três níveis: um mecanismo de reconhecimento HIDS, outro baseado NIDS e o terceiro é um módulo administrador. É um bom exemplo de um sistema IDS híbrido.

As ferramentas de domínio público não possuem suporte e acabam tendo o processo de instalação e manutenção comprometido, pois não existem empresas patrocinando o desenvolvimento. Mas é válido avaliar estas aplicações para entender como funciona a tecnologia de IDS:

- **Shadow** – Utiliza chamadas a sensores (que ficam espalhados pela rede) e estações de análise. A filosofia do Shadow é não emitir alertas, simplesmente são criados arquivos de registro. Utiliza a biblioteca *libpcap* para prover uma capacidade de *sniffer* básica;

- **NFR** (*Network Flight Recorder*) – IDS disponível em uma versão comercial e outra de domínio público (mais antiga). Utiliza a biblioteca *libpcap* para extrair de forma aleatória e passiva os pacotes da rede para análise. Pode agir fora de um *firewall* para descobrir ameaças em pontos mais distantes da rede. Possui uma linguagem de programação completa que permite desenvolver scripts de análise mais completos;
- **Tripwire** – Como o NFR, existe uma versão comercial e outra de domínio público (códigos fontes disponíveis para Unix e Linux). O Tripwire trabalha de forma diferente de outros IDS, ele verifica os arquivos de sistema em busca de mudanças. É feito um *checksum* entre o arquivo de sistema e as informações que estão armazenadas num sistema seguro. Com o Tripwire é possível restaurar os arquivos modificados. Trabalha com o conceito de assinatura criptográfica.

3.5.5 - Produtos GOTS (*Government Off-the-Shelf*) – Produtos não disponíveis comercialmente

Enquanto as empresas utilizam IDS visando proteger a sua rede para obter lucro (não perdendo dados e produtos) e/ou não ser responsável por uma invasão que possa ocorrer com seus parceiros comerciais ou acionistas, os órgãos governamentais também estão interessados em proteger a sua rede, mas com um outro foco e objetivo: A proteção da soberania nacional [ALL99].

No seminário "*Detection of Malicious Code, Intrusions, and Anomalous Activity*" realizado em 1999 e patrocinado pelo Departamento de Energia, Conselho de Segurança Nacional e Departamento de Ciência e Políticas Tecnológicas (USA) e que teve a participação de especialistas ligados a setores do governo e comércio, chegou-se a algumas considerações:

- Qualquer atacante que tenha o apoio de uma nação terá maiores recursos e ferramentas que um atacante comum;
- Um IDS para fins governamentais deve ter a capacidade de avaliar e armazenar a maior quantidade de dados possíveis para análise e uso futuro – identificação de intenção;
- O objetivo de um IDS numa empresa é barrar o ataque e evitar qualquer prejuízo, para a nação é descobrir as respostas para as questões: Quem ? O que ? Por quê ? Quando ? E como ? Um IDS comercial não tem a necessidade de responder a estas questões – e para sua própria sobrevivência, dificilmente virá a responder estas

questões;

- Qualquer pessoa pode adquirir um IDS comercial, neste caso um atacante sabendo que um governo está utilizando este IDS pode adquiri-lo e descobrir como derrotá-lo.

As necessidades de um governo sempre serão superiores aos que as ferramentas atuais proporcionam, para essas necessidades é essencial que o governo continue a desenvolver e patrocinar IDS GOTS.

3.6 Conclusão

Permanece em aberto se a tecnologia de descoberta de intrusão pode cumprir a promessa de identificar ataques com precisão, são muitas as propostas, mas poucos resultados na prática [ALL99], esta afirmação pode ser constada pela quantidade de trabalhos propostos para a detecção de intrusão. A tecnologia atual utiliza um universo pequeno de técnicas para detectar ataques e intrusão. Como a tecnologia de intrusão está evoluindo mais rapidamente que a de detecção, uma ferramenta de IDS deve possuir algumas características importantes [ALL99], entre elas:

- Deve rodar continuamente sem interação humana e deve ser segura o suficiente de para permitir sua operação em *background*, mas deve ser fácil compreensão e operação;
- Deve ter tolerância à falhas, de forma a não ser afetada por uma falha do sistema, ou seja, sua base de conhecimento não deve ser perdida quando o sistema for reinicializado;
- Deve resistir a tentativas de mudança (subversão) de sua base, ou seja, deve monitorar a si próprio de forma a garantir sua própria segurança;
- Dever ter o mínimo de impacto no funcionamento do sistema;
- Deve detectar mudanças no funcionamento normal;
- Deve ser de fácil configuração, cada sistema possui padrões diferentes e a ferramenta de IDS deve ser adaptada de forma fácil aos diversos padrões;
- Deve cobrir as mudanças do sistema durante o tempo, como no caso de uma nova aplicação que comece a fazer parte do sistema;
- Deve ser difícil de ser enganada.

Além dos tópicos citados anteriormente, um IDS deve estar protegido e inacessível a

ataques e invasores.

Somente a utilização de um IDS não garante a segurança computacional do sistema, sendo necessária à utilização de outras tecnologias e procedimentos, entre as quais estão a utilização de *firewalls*, utilização de antivírus atualizados, instalação de correções de segurança, políticas de segurança bem documentadas e acessíveis a todos os funcionários, treinamento adequado, configurações corretas e boa administração dos sistemas operacionais, utilização de criptografia e certificação digital para a proteção dos dados [STA98].

Capítulo 4

Chamadas de Sistema e Segurança

Uma preocupação que surge na grande maioria dos projetos de sistemas operacionais é a implementação de mecanismos de proteção ao *kernel* do sistema e de acesso aos seus serviços. Caso uma aplicação realize uma operação que o danifique, todo o sistema poderá ficar comprometido e inoperante. A forma encontrada para a proteção do sistema, foi a criação das chamadas de sistema (*system calls* ou *syscalls*) para fornecerem uma interface entre os processos e o núcleo do sistema operacional.

As instruções que têm o poder de comprometer o sistema são conhecidas como instruções privilegiadas, enquanto as instruções não-privilegiadas são as que não oferecem perigo ao sistema.

Para que uma aplicação possa executar uma instrução privilegiada, o processador implementa o mecanismo de modos de acesso. Existem basicamente dois modos de acesso implementados pelo processador: modo usuário e modo *kernel*.

Quando o processador trabalha no modo usuário, uma aplicação só pode executar instruções não-privilegiadas, tendo acesso a um número reduzido de instruções, enquanto no modo *kernel* a aplicação pode ter acesso ao conjunto total de instruções do processador.

O modo de acesso de uma aplicação é determinado por um conjunto de *bits*, localizado em um registrador especial do processador, que indica o modo de acesso corrente. Através desse registrador, o *hardware* verifica se a instrução pode ou não ser executada pela aplicação.

A melhor maneira de controlar o acesso às instruções privilegiadas é permitir que apenas o sistema operacional tenha acesso a elas. Sempre que uma aplicação necessita de um

serviço que incorra em risco para o sistema, a solicitação é feita através de uma chamada de sistema. A chamada de sistema altera o modo de acesso do processador para um modo mais privilegiado (modo *kernel*). Ao término da rotina do sistema, o modo de acesso é retornado para o modo usuário. Caso um programa tente executar uma instrução privilegiada, sem o processador estar no modo *kernel*, uma exceção é gerada e o programa é encerrado.

O núcleo do sistema operacional sempre é executado em modo *kernel*, pois deve possuir capacidade de gerenciar e compartilhar todos os seus recursos, solucionando, em diversos níveis, os problemas de acesso às instruções privilegiadas.

Este capítulo discute a importância das chamadas de sistema como porta de entrada de ataques ao *kernel* do sistema operacional e como as chamadas de sistema podem ser utilizadas para identificar e evitar ataques ao próprio sistema operacional.

4.1 Conceituação

As chamadas de sistema (*system calls*) constituem a interface entre os processos e o sistema operacional. Essas chamadas estão geralmente disponíveis como instruções em linguagem *assembly* e, em geral, são listadas nos manuais usados pelos programadores [SIL00a, SIL00b]. No sistema Linux, as chamadas de sistema estão listadas nos arquivos `/usr/include/sys/syscall.h` ou `/usr/include/bits/syscall.h`.

A aplicação, quando deseja solicitar algum serviço do sistema, realiza uma chamada a uma de suas rotinas (ou serviços) através de chamadas de sistema, que são a porta de entrada para se ter acesso ao *kernel* do sistema operacional. Para cada serviço existe uma chamada de sistema associada e cada sistema operacional tem o seu próprio conjunto de chamadas, com nomes, parâmetros e formas de ativação específicos.

Através dos parâmetros fornecidos na chamada de sistema, a solicitação é processada e uma resposta é retornada à aplicação, em um dos parâmetros fornecidos na chamada ou como retorno da chamada de sistema. O mecanismo de ativação e comunicação entre a aplicação e o sistema é semelhante ao mecanismo implementado quando um programa modularizado ativa um dos seus procedimentos ou funções.

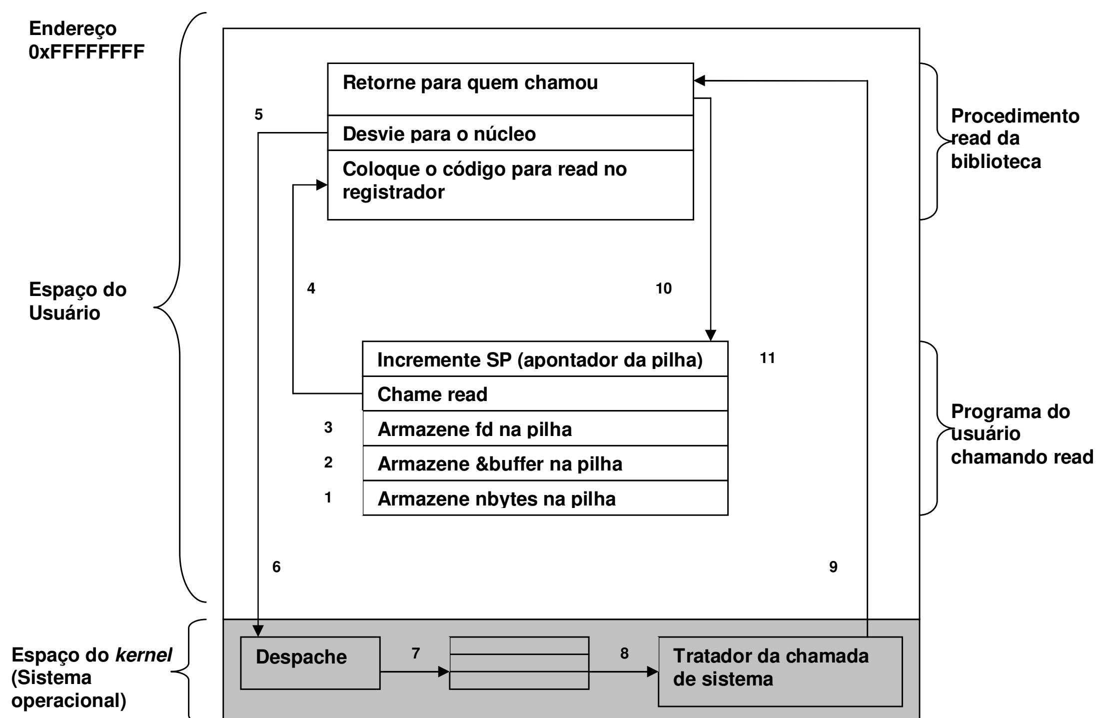

Figura 4.1 – 11 passos para fazer uma chamada read(arq, buffer, nbytes) [TAN03]

As chamadas de sistema são realizadas em uma série de passos. A figura 4.1 exemplifica o processo (utilizando a chamada de sistema `read`) realizado durante uma chamada de sistema:

- O programa que está chamando armazena os parâmetros na pilha (passos 1 a 3);
- Ocorre a chamada real ao procedimento (4);
- Colocação do número de chamada ao sistema em um local esperado pelo sistema operacional, por exemplo, um registrador (5);
- Mudança do modo usuário para o modo *kernel* (6);
- Verificação da chamada e despache do procedimento correto (7);
- Execução do procedimento de tratamento da chamada (8);
- Retorno ao modo usuário (9) e ao programa do usuário (10);
- Limpeza da pilha (11).

As chamadas de sistema podem ser agrupadas em cinco categorias principais: controle de processos, manipulação de arquivos, manipulação de dispositivos, manutenção de informações e comunicações [SIL00a, SIL00b]. A tabela 4.1 ilustra as chamadas de sistema comuns a todos os sistemas operacionais.

Tabela 4.1 – Chamadas de Sistema comuns [SIL00a, SIL00b]

Grupo	**Chamadas de Sistema**
Controle de Processos	*end, abort, load, execute, create process, terminate process, get process attributes, set process attributes, wait for time, wait event, signal event, allocate memory, free memory*
Gerência de Arquivos	*create file, delete file, open, close, read, write, reposition, get file attributes, set file attributes*
Gerência de Dispositivos	*request device, release device, read, write, reposition, get device attributes, set device attributes, logically attach device, logically detach device*
Manutenção de Informações	*get time, get date, set time, set date, get system data, set system data, get process attributes, get file attributes, get device attributes, set process attributes, set file attributes, set device attributes*
Comunicações	*create communication connection, delete communication connection, send message, receive message, transfer status information, attach remote devices, detach remote devices*

4.2 Ataques a Chamadas de Sistema

Devido à importância das chamadas de sistema para o sistema operacional e para as aplicações de usuário, as chamadas de sistema acabam se tornando um dos pontos mais atacados num sistema computacional para se obter acesso privilegiado ao sistema.

É comum ocorrer vulnerabilidades em chamadas de sistema de sistemas operacionais específicos, estas vulnerabilidades ocorrem normalmente por erros de programação e são normalmente corrigidas através da liberação de atualizações (*patch*). Os principais problemas de segurança atuais, envolvendo as chamadas de sistema, são: as vulnerabilidades de *buffer overflow, DoS* (*Denial of Service* – Negação de Serviço) e a instalação de *rootkits*.

4.2.1 Denial of Service

Um ataque do tipo *DoS* basicamente visa indisponibilizar os serviços oferecidos por algum servidor de rede, como *WebServer*, *mail* ou DNS (*Domain Name Service*). Indisponibilizar pode significar retirar totalmente o servidor de operação ou apenas deixá-lo lento, a ponto do cliente abandonar o serviço devido ao tempo de resposta. Um ataque *DoS* não implica no servidor ser invadido, ou seja, as informações contidas no servidor não correm necessariamente algum risco.

Normalmente, os ataques *DoS* atuam nas fraquezas dos protocolos TCP/IP, sendo possível de ser implementado em qualquer dispositivo que utilize este protocolo, como servidores, clientes e roteadores. Um exemplo comum de ataque *DoS* é *TCP SYN Attack* [SCH97]. Um ataque *Distributed DoS* [LAU00] nada mais é que um ataque *DoS* em larga escala, utilizando uma dezena, centena ou milhares de sistemas ao mesmo tempo no ataque de um ou mais alvos. Este tipo de ataque é considerado como sendo de alto risco e de difícil defesa.

O *fork bomb* é outro exemplo de ataque *DoS*. Este ataque visa indisponibilizar o sistema operacional através da criação de um grande número de processos (muito além do que o sistema operacional consegue gerenciar). A tabela 4.2 apresenta um exemplo de código de *fork bomb* para o sistema operacional Unix.

Tabela 4.2 – Exemplo de *fork bomb*

```
#include <unistd.h>

void main(void)
{
   while(1) fork();
}
```

A chamada de sistema `fork` do sistema Unix, possibilita a criação de novos processos. O código da tabela 4.2 utiliza o `fork` para duplicar-se indefinidamente, até que o sistema operacional fique indisponível por não mais conseguir gerenciar os inúmeros processos criados.

Os ataques de *DoS* podem ocorrer através de vulnerabilidades na programação de chamadas de sistema de alguns sistemas operacionais. Exemplos de chamadas de sistema que possuem alguma vulnerabilidade e que poderia ser utilizada para ataques de *DoS* podem ser

encontrados em [IBM99, SAF00, SAF01, CER02, STA03, SEC03a, SEC03b, SEC03c, STA04].

4.2.2 Buffer Overflow

Os ataques de *buffer overflow* foram a principal vulnerabilidade de segurança nos últimos 10 anos, sendo responsáveis pela maioria dos ataques remotos (através da Internet) que permitiam de forma anônima o acesso indevido a um sistema operacional [COW00].

Ataques explorando a vulnerabilidade de *buffer overflow* tornaram-se os preferidos dos atacantes, pela particularidade de poder injetar e executar códigos maliciosos. Um programa atacado permite que um atacante execute e consiga o acesso ao sistema operacional com os mesmos privilégios do usuário que estava rodando o programa. Normalmente, os ataques são direcionados a programas que executam sob o usuário administrador do sistema operacional. Nos sistemas Unix, os programas em execução com o usuário `root` são atacados para a execução de um código similar ao `exec(sh)`, que permite ganhar o *shell* do usuário `root`.

A vulnerabilidade de *buffer overflow* é ocasionada pelo fato das chamadas de sistema de alguns sistemas operacionais não tratarem adequadamente alguns aspectos do gerenciamento de memória, delegando esta função para o programador da aplicação [COW00]. Cabe ao programador garantir o correto gerenciamento de memória de sua aplicação no sistema. Esta é uma característica da linguagem de programação C, usada no desenvolvimento dos sistemas operacionais Unix e Linux e na maior parte suas aplicações, tornando estes sistemas os alvos principais para este tipo de vulnerabilidade.

O primeiro grande incidente de segurança da Internet – o *Morris Worm*, em 1988 – utiliza técnicas de *buffer overflow* no programa `fingerd`. Outros exemplos de vulnerabilidades de *buffer overflow* podem ser encontrados em [CER03a, CER03b, CER03c].

4.2.3 Rootkits

O termo *rootkit* vem designar uma série de ferramentas utilizadas por um invasor para modificar ou ocultar sua presença em um sistema invadido. A idéia inicial é que uma série de programas disfarçados em arquivos do sistema pudesse realizar tarefas de roubo de informações, possibilidade de acesso não autorizado a qualquer momento e em caso de necessidade, desativação da máquina hospedeira dos mesmos, para que o invasor não possa ser detectado.

Rootkits são pacotes de software, que após a instalação permitem que atacantes ganhem acesso privilegiado ao sistema. Rootkits tradicionais, normalmente, substituem os binários dos comandos `ls`, `ps` e `netstat` para esconder os ataques aos arquivos, processos e conexões de rede. Estes *rootkits* são facilmente detectáveis através da checagem de integridade dos códigos binários.

Rootkits de *kernel* não substituem binários dos comandos, mas subvertem o *kernel* do sistema. Por exemplo, o comando `ps` lê as informações do diretório `/proc` (sistema de arquivos `procfs`). Um *rootkit* de *kernel* subverte o *kernel* para esconder os processos específicos do `procfs`, desta forma não é necessário alterar o binário do comando `ps`. Estes *rootkits* também redirecionam a execução das chamadas de sistema através da alteração da tabela de chamadas de sistema do sistema. As chamadas de sistema podem ser alteradas para esconder informações do sistema, ou para retornar falsos dados quando ocorrer o processo de leitura/escrita nos *devices* do sistema operacional [ZON01].

4.3 Detecção de Intrusão por Análise de Chamadas de Sistema

A detecção de intrusão por análise de chamadas de sistema é uma das técnicas com origem na análise por anomalia proposta por [DEN87]. Os trabalhos [FOR96, HOF98] descrevem técnicas para detecção de intrusão a partir da análise da execução de chamadas de sistema de um processo. O algoritmo para geração de uma base de dados é extremamente simples, sendo realizado a partir do rastreamento das execuções de chamadas de sistema de um processo em particular. A base de dados é montada em seqüências de chamadas de sistema de comprimento k, que são capturadas durante toda a vida do processo. O valor de k indica a quantidade de chamadas de sistema que cada seqüência terá.

A construção de um conjunto de dados válidos pode ser ilustrada no exemplo a seguir. Considerando a seguinte seqüência de chamadas de sistema (sem seus parâmetros):

`open, read, mmap, mmap, open, read, mmap`

Pode-se derivar as seguintes seqüências de chamadas com comprimento $k = 3$:

S_1 = { `open, read, mmap` }

S_2 = { `read, mmap, mmap` }

S_3 = { `mmap, mmap, open` }

S_4 = { `mmap, open, read` }

S_5 = { `open, read, mmap` }

De acordo com o algoritmo, uma suspeita de intrusão será levantada caso seja detectada uma seqüência de chamadas desconhecida S_d, não registrada na base de seqüências conhecidas, como o exemplo S_d = `{ open, read, read }`.

A maior dificuldade na detecção de intrusão por análise de chamadas de sistema, através deste algoritmo, reside na criação da base de seqüências conhecidas. O conteúdo desta base irá depender da versão do sistema operacional e os aplicativos sob análise: diferentes versões do mesmo aplicativo irão provavelmente gerar seqüências distintas de chamadas de sistema. Esta limitação exige que a base histórica seja a mais completa e confiável possível para minimizar as ocorrências de falsos positivos. Outro aspecto do algoritmo, que dificulta a criação da base histórica é controle das chamadas de sistema realizadas durante o ciclo de vida de processos resultantes do mesmo executável (comando), sendo necessário diferenciar as seqüências de chamadas de sistema de cada processo, esta diferenciação pode ocorrer através do número do processo dentro do sistema operacional (no ambiente Unix/Linux, *Process Identification – PID*).

Outro aspecto influente na eficiência deste algoritmo é a escolha do comprimento do *k* para as seqüências conhecidas. De acordo com [HOF98], valores de *k* pequenos gerarão bases compactas, de fácil manejo, mas com menor garantia de detecção de intrusões. Valores maiores de *k* gerarão bases que irão representar mais fielmente o comportamento dos processos, mas que serão maiores e mais difíceis de gerenciar. Alguns autores propõem o uso de uma máquina de estados para registrar seqüências conhecidas, em substituição à base de seqüências [WAR99]. Esta abordagem teria a vantagem de registrar integralmente o comportamento conhecido dos processos, mas sua implementação é mais complexa e sujeita à influência da versão do *software* utilizado.

Outros autores descrevem a utilização da análise de seqüências de chamadas de sistema associados a outras métricas, tais como métodos estatísticos [HEL97], predição através da utilização de árvores de decisão [LEE97, LEE98], redes neurais [GHO99] para a modelagem dos dados ou a utilização dos modelos de *Markov* [YE00, ESK01]. Estes trabalhos comprovaram a eficiência da análise de chamadas de sistema como um método confiável para a detecção de intrusão.

4.4 Classificação de Chamadas de Sistema

Como visto anteriormente, a detecção de intrusão por análise de chamadas de sistema

tem se mostrado um excelente método para detectar invasões a sistemas de computadores. Embora uma invasão possa ser detectada pelos mecanismos de IDS e combatida com outros métodos, uma análise das chamadas de sistema é necessária para projetar e implementar um sistema seguro.

No trabalho [BER00, BER02], é apresentado uma proposta para tornar um sistema operacional mais seguro. Esta proposta baseia-se no controle das execuções das chamadas de sistema no sistema operacional, portanto uma classificação, sob o aspecto da segurança computacional, das chamadas de sistema tornou-se necessária.

Naquele trabalho, as chamadas de sistema foram classificadas de acordo com sua área de atuação (tabela 4.3) e nível de ameaça. As chamadas de sistema classificadas com nível de ameaça 1 podem ser utilizadas para ganhar o acesso total ao sistema operacional, no nível 2 estão as chamadas de sistema que podem ser utilizadas para ataques de *Denial of Service* (*DoS*), no nível 3 estão as chamadas de sistema que podem ser usadas para subverter processos e no nível 4 estão as chamadas de sistema consideradas inofensivas sob o aspecto de segurança computacional.

Tabela 4.3 – Categoria de Chamadas de Sistema

Grupo	**Funcionalidade**
I	Acesso ao sistema de arquivos e *device drivers*
II	Gerência de processos
III	Gerência de módulos
IV	Gerência de Memória
V	Tempo
VI	Comunicação
VII	Informações de Sistema
VIII	Reservado
IX	Não implementado

Através do controle das execuções das chamadas de sistema, o sistema de segurança proposto impede o comprometimento do sistema operacional, pois a execução das chamadas de sistema críticas podem ser acompanhadas e canceladas. A tabela 4.4 demonstra a classificação das chamadas de sistema por grupo e nível de ameaça.

No protótipo implementado, denominado REMUS, o *kernel* do sistema Linux na versão 2.2 foi alterado para verificar e controlar a execução das chamadas de sistema. Através da utilização de um monitor, todas as chamadas de sistema são verificadas numa base de controle de acesso (ACD – *Access Control Database*). Toda chamada as chamadas de sistema são auditadas e o *kernel*, após verificação no ACD, permite ou nega a execução de uma chamada de sistema.

Com esta abordagem o *kernel* pode recusar a execução de chamadas de sistema oriundas de processos considerados inofensivos, mas que podem estar sofrendo ataques, como *buffer overflow*, visando obter o acesso pleno ao sistema operacional.

Tabela 4.4 – Categoria de Chamadas de Sistema Classificadas por nível de ameaça

<table>
<tr><th>Ameaça</th><th>Grupo</th><th>Chamadas de Sistema</th></tr>
<tr><td rowspan="3">1</td><td>I</td><td>open, link, unlink, chmod, lchown, rename, fchown, chown, mknod, mount, symlink, fchmod</td></tr>
<tr><td>II</td><td>execve, setgid, setreuid, setregid, setgroups, setfsuid, setfsgid, setresuid, setresgid, setuid</td></tr>
<tr><td>III</td><td>init_module</td></tr>
<tr><td rowspan="7">2</td><td>I</td><td>creat, umount, mkdir, rmdir, umount2, ioctl, nfsservctl, truncate, ftruncate, quotactl, afs syscall, dup2, flock</td></tr>
<tr><td>II</td><td>fork, brk, kill, setrlimit, reboot, setpriority, ioperm, iopl, clone, modify ldt, adjtimex, sched setparam, vfork, vhangup, sched setscheduler, vm86, vm86old</td></tr>
<tr><td>III</td><td>delete_module</td></tr>
<tr><td>IV</td><td>swapon, swapoff, mlock, mlockall</td></tr>
<tr><td>V</td><td>stime, settimeofday, nice</td></tr>
<tr><td>VI</td><td>socketcall, ipc</td></tr>
<tr><td>VII</td><td>sethostname, syslog, setdomainname, sysctl</td></tr>
<tr><td>3</td><td>I</td><td>read, write, close, chdir, lseek, dup, fcntl, umask, chroot, select, fsync, fchdir, llseek, newselect, readv, writev, poll, pread, pwrite, sendfile, putpmsg, utime</td></tr>
</table>

3	II	*exit, waitpid, ptrace, signal, setpgid, setsid, sigaction, ssetmask, sigsuspend, sigpending, uselib, wait4, sigreturn, sigprocmask, personality, capset, rt_sigreturn, rt_sigaction, rt_sigprocmask, rt_sigpending, rt_sigtimedwait, rt_sigqueueinfo, rt_sigsuspend, sched_yield, prctl*
	IV	*mmap, munmap, mprotect, msync, munlock, munlockall, mremap*
	V	*pause, setitimer, nanosleep*
4	I	*oldstat, oldfstat, access, sync, pipe, ustat, oldlstat, readlink, readdir, statfs, fstatfs, stat, getpmsg, lstat, fstat, olduname, bdflush, sysfs, getdents, fdatasync*
	II	*getpid, getppid, getuid, getgid, geteuid, getegid, acct, getpgrp, sgetmask, getrlimit, getrusage, getgroups, getpriority, sched_getscheduler, sched_getparam, sched_get_priority_max, sched_get_priority_min, sched_rr_get_interval, capget, getpgid, getsid, getcwd, getresgid, getresuid*
	III	*get_kernel_syms, create_module, query_module*
	V	*times, time, gettimeofday, getitimer*
	VII	*sysinfo, uname*
	VIII	*idle*
	IX	*break, ftime, mpx, stty, prof, ulimit, gtty, lock, profil*

4.5 Conclusão

As chamadas de sistema são disponibilizadas para fornecerem uma interface entre um processo e o sistema operacional. O uso indevido de uma chamada de sistema pode afetar o sistema operacional. Os ataques às chamadas de sistema podem comprometer a segurança de um sistema operacional e disponibilizar acesso privilegiado a pessoas não autorizadas. O acompanhamento e controle da execução das chamadas de sistema das aplicações e do próprio *kernel* podem deletar e neutralizar possíveis ataques.

Capítulo 5

Proteção de Detectores de Intrusão Usando Máquinas Virtuais

A garantia da segurança no funcionamento de um sistema torna-se uma tarefa cada vez mais complexa, devido à multiplicação das formas de ataque e atacantes [ALL99]. Os sistemas de detecção de intrusão atuais apresentam várias deficiências, conforme apresentado no capítulo 3. Neste capítulo é proposto um modelo para a detecção de intrusão em sistemas de produção, aproveitando os benefícios da utilização de máquinas virtuais nestes ambientes.

5.1 Problema

O principal problema relacionado a sistemas HIDS reside no fato que o IDS reside na própria máquina a monitorar, ou seja, o sistema também deve monitorar (e proteger) a si próprio. Assim, um sistema HIDS torna-se inútil no caso de um ataque bem sucedido, pois as informações oriundas deste sistema não podem mais ser consideradas confiáveis [WAD00], pois o IDS pode ser subvertido ou mesmo desativado por um invasor. O uso de máquinas virtuais permite solucionar esse problema, constituindo uma alternativa interessante para a implantação de sistemas do tipo HIDS, conforme apresentado na proposta a seguir.

5.2 Proposta

A proposta deste trabalho é uma arquitetura para a aplicação de sistemas HIDS de forma robusta e confiável. Isso é obtido através da execução dos processos de aplicação a monitorar em máquinas virtuais e a implantação dos sistemas de detecção e resposta a intrusão fora da máquina virtual (portanto fora do alcance de eventuais intrusos).

A figura 5.1 ilustra o modelo genérico da arquitetura de segurança proposta. Através de interfaces é possível extrair informações da máquina virtual e enviar para o sistema anfitrião (1). Uma aplicação no sistema anfitrião efetua a análise das informações coletadas (2), e, caso necessário, uma intervenção pode ser feita diretamente na máquina virtual em execução (3). Todo o ciclo ocorre de forma transparente para a máquina virtual e os processos nela contidos.

A interação entre o sistema convidado e o IDS externo é feita através do monitor de máquina virtual. São definidos dois tipos de interações:

- **Coleta** – dados do sistema convidado são fornecidos pelo monitor de máquina virtual para análise externa; por exemplo, podem ser disponibilizadas as chamadas de sistema dos processos convidados;
- **Ação** – o sistema de detecção externo pode comandar ações sobre o sistema convidado, como encerrar processos, remover arquivos, encerrar conexões de rede, etc, em resposta a ataques detectados.

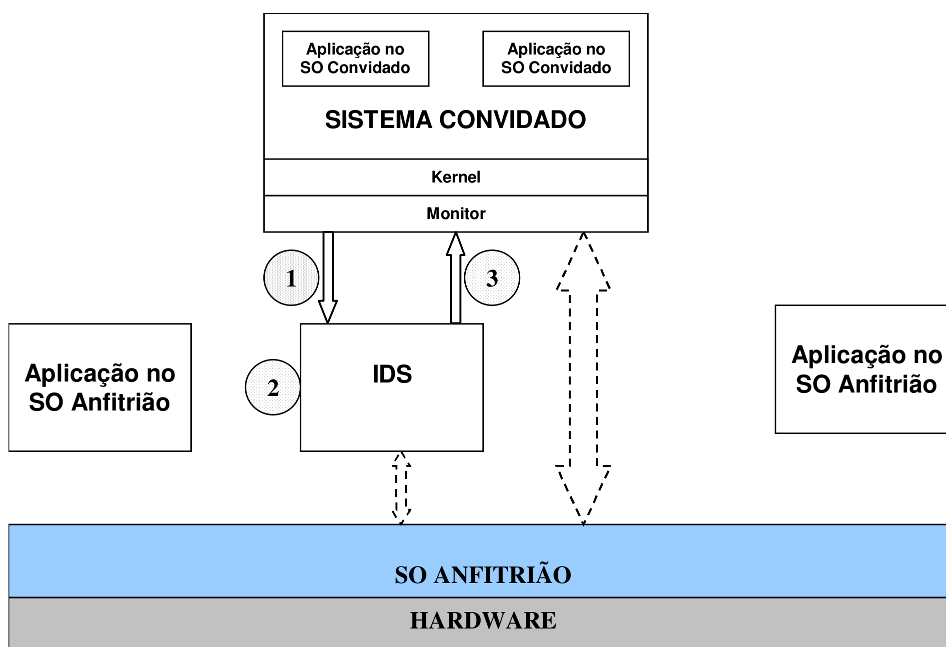

Figura 5.1 – Modelo Genérico

5.3 Funcionamento da Proposta

Como visto no capítulo 4, a detecção de intrusão por análise de seqüências de

chamadas de sistema é uma abordagem confiável e viável. Durante o ciclo de vida de um processo, várias chamadas de sistema são executadas pelo sistema operacional, em VMs do Tipo II é possível interceptar estas seqüências antes da sua execução e utilizar os mecanismos já existentes para a verificação de uma possível tentativa de invasão ao sistema convidado.

O IDS pode operar em dois modos: *aprendizado* e *monitoração*. Ao executar o sistema no modo **aprendizado**, os processos em execução na máquina virtual e seus usuários são registrados como processos e usuários autorizados. O sistema também pode registrar o fluxo das chamadas de sistema de processos específicos. O modo *aprendizado* permite, portanto, registrar o comportamento "normal" do sistema, coletando dados que serão essenciais para o processo de detecção de intrusão.

No modo **monitoração**, o IDS recebe continuamente informações do monitor de máquina virtual e as compara com os dados previamente armazenados. No protótipo atual são analisadas as seqüências de chamadas de sistema de acordo com o algoritmo proposto por [FOR96, HOF98], mas qualquer outra abordagem para detecção de intrusão por análise de chamadas de sistema poderia ser utilizada. Caso uma seqüência de chamadas de sistema não esteja registrada para um determinado processo, uma situação anormal é sinalizada e aquele processo é declarado *suspeito*.

A figura 5.2 ilustra o funcionamento do IDS, sua capacidade de reação e o seu relacionamento e do monitor com o sistema anfitrião. Após a recepção das informações, é realizada uma análise na base histórica para detectar anomalias no comportamento, um *status* é enviado para o mecanismo de verificação de acesso (ACL) para a verificação das permissões do usuário e uma decisão é tomada. A decisão é enviada para o monitor que providencia as ações desejadas (bloqueio da conexão de rede ou suspensão da execução de um processo são exemplos das ações que podem ser tomadas).

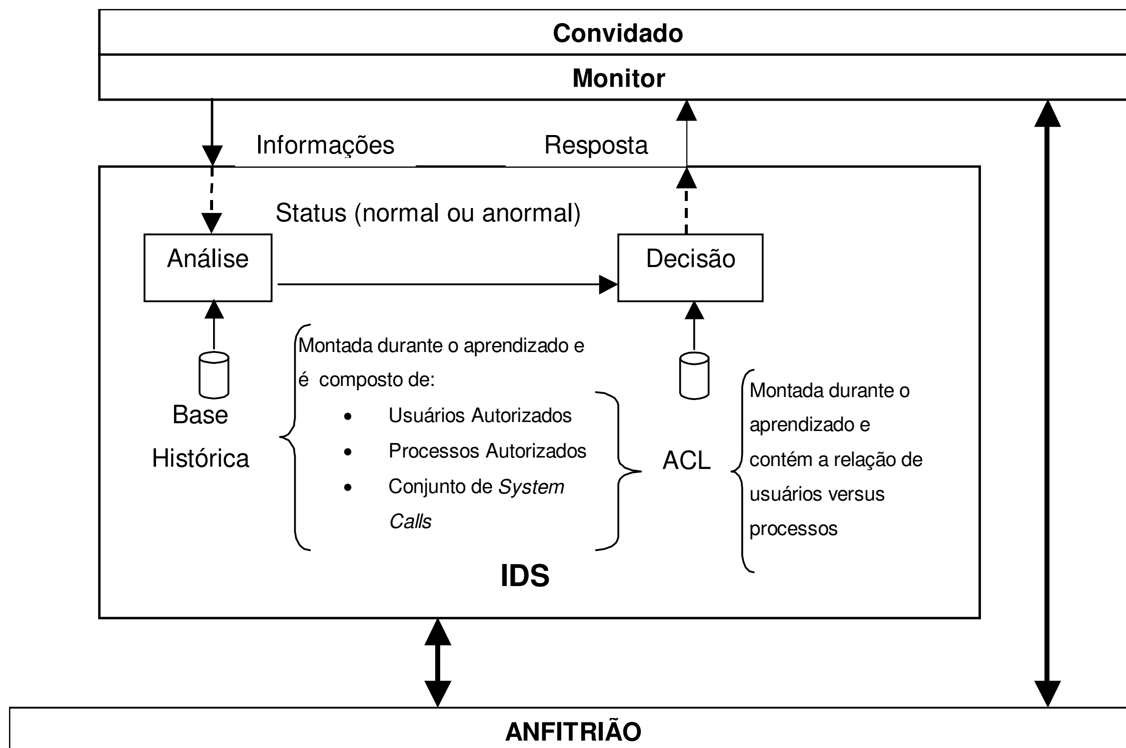

Figura 5.2 – Funcionamento do IDS

5.3.1 O Processo de Aprendizado

O processo de **aprendizado** é responsável por montar a base histórica de utilização do sistema. O IDS recebe as informações – usuários que utilizaram o sistema e nome dos processos executados e seqüências de chamadas de sistema dos processos – e registra essas informações para uso posterior. As informações são coletadas durante todo o ciclo de vida da máquina virtual (figura 5.3).

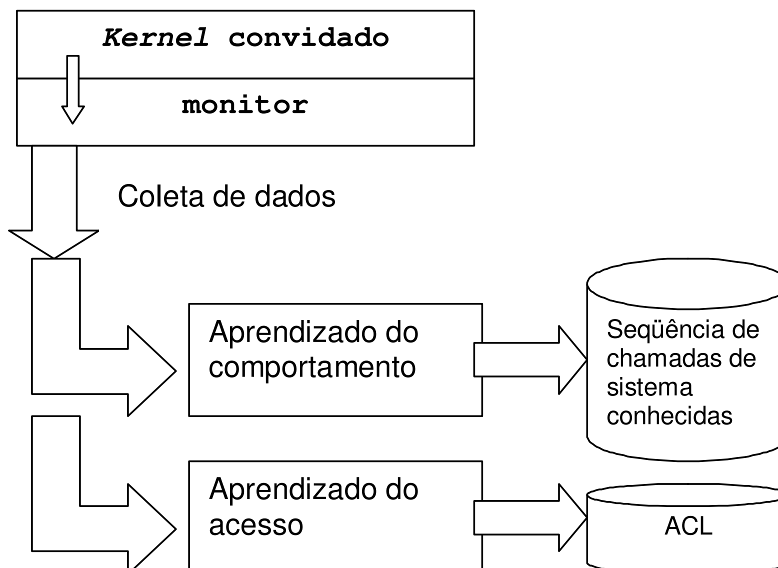

Figura 5.3 – IDS em modo Aprendizado

Para as seqüências de chamadas de sistema coletadas de cada processo, são geradas amostras de tamanho $k = 3$ e armazenadas na máquina real. Foi escolhido $k = 3$ pela simplicidade e performance do algoritmo para o gerenciamento (inclusão/consulta) das seqüências geradas. A tabela 5.1 apresenta a seqüência de chamadas de sistema realizadas em uma execução do comando `who` (a listagem incluindo os parâmetros e os resultados da execução se encontram no apêndice A). A tabela 5.2 ilustra o mesmo conjunto de chamadas de sistema agrupadas em amostras de tamanho 3. Estas seqüências serão utilizadas posteriormente para detectar modificações no comportamento durante o ciclo de vida do processo, que podem caracterizar um ataque ou a substituição de um código binário. O monitor retorna as informações de chamadas de sistema na forma de números (listadas no arquivo `/include/asm/unistd.h` do código fonte do *kernel* do Linux), para fins de ilustração as tabelas 5.1 e 5.2 (na tabela 5.2 não consta as seqüências de chamadas de sistema repetidas) foram criadas utilizando os respectivos nomes das chamadas de sistema.

Tabela 5.1 – Seqüência de chamadas de sistema sem parâmetros do comando `who`

```
execve, uname, brk, old_mmap, open, open, fstat64, old_mmap, close, open,
read, fstat64, old_mmap, old_mmap, old_mmap, close, munmap, open, brk,
brk, brk, brk, open, fstat64, mmap2, read, read, close, munmap, open,
fstat64, mmap2, close, open, fstat64, mmap2, close, open, fstat64, mmap2,
close, open, fstat64, mmap2, close, open, fstat64, mmap2, close, open,
```

```
fstat64, mmap2, close, open, fstat64, close, open, fstat64, mmap2, close,
open, fstat64, mmap2, close, open, fstat64, mmap2, close, open, fstat64,
mmap2, close, open, fstat64, mmap2, close, open, fstat64, mmap2, close,
access, open, open, fcntl64, fcntl64, _llseek, alarm,
rt_sigaction, alarm, fcntl64, read, fcntl64, alarm, rt_sigaction, alarm,
…(16 vezes a linha acima)
rt_sigaction, alarm, fcntl64, read, fcntl64, alarm, rt_sigaction, close,
stat64, time, open, fstat64, mmap2, read, close, munmap, fstat64, mmap2,
stat64, open, fstat64, mmap2, close, open, fstat64, mmap2, close, stat64,
stat64, stat64, stat64, stat64, stat64, write, close, munmap, exit_group
```

Tabela 5.2 – Conjunto de chamadas de sistema agrupadas

```
{execve,uname,brk}, {uname,brk,old_mmap},{brk,old_mmap,open},
{old_mmap,open,open}, {open,open,fstat64}, {open,fstat64,old_mmap},
{fstat64,old_mmap,close}, {old_mmap,close,open}, {close,open,read},
{open,read,fstat64}, {read,fstat64,old_mmap}, {fstat64,old_mmap,old_mmap},
{old_mmap,old_mmap,old_mmap}, {old_mmap,old_mmap,close},
{old_mmap,close,munmap}, {close,munmap,open}, {munmap,open,brk},
{open,brk,brk}, {brk,brk,brk}, {brk,brk,open}, {brk,open,fstat64},
{open,fstat64,mmap2}, {fstat64,mmap2,read}, {mmap2,read,read},
{read,read,close}, {read,close,munmap}, {munmap,open,fstat64},
{fstat64,mmap2,close}, {mmap2,close,open}, {close,open,fstat64},
{open,fstat64,close}, {fstat64,close,open}, {mmap2,close,access},
{close,access,open}, {access,open,open}, {open,open,fcntl64},
{open,fcntl64,fcntl64}, {fcntl64,fcntl64,_llseek},
{fcntl64,_llseek,alarm}, {_llseek,alarm,rt_sigaction},
{alarm,rt_sigaction,alarm}, {rt_sigaction,alarm,fcntl64},
{alarm,fcntl64,read}, {fcntl64,read,fcntl64}, {read,fcntl64,alarm},
{fcntl64,alarm,rt_sigaction}, {rt_sigaction,alarm,rt_sigaction},
{alarm,rt_sigaction,close}, {rt_sigaction,close,stat64},
{close,stat64,time}, {stat64,time,open}, {time,open,fstat64},
{mmap2,read,close}, {close,munmap,fstat64}, {munmap,fstat64,mmap2},
{fstat64,mmap2,stat64}, {mmap2,stat64,open}, {stat64,open,fstat64},
{mmap2,close,stat64}, {close,stat64,stat64}, {stat64,stat64,stat64},
{stat64,stat64,write}, {stat64,write,close}, {write,close,munmap},
{close,munmap,exit_group}
```

No modo de aprendizado, o IDS também cria as ACLs dos processos e usuários

autorizados. Na medida em que o sistema vai sendo utilizado, o IDS registra o comportamento de utilização do sistema, no encerramento do modo aprendizado, é gerado um arquivo que contém os processos utilizados e seus respectivos usuários. A tabela 5.3 ilustra o formato do arquivo gerado.

Tabela 5.3 – Relação usuário *versus* processo

...
root:/sbin/init
root:/sbin/shutdown
root:/usr/bin/find
marcos:/bin/cp
marcos:/usr/bin/find
marcos:/usr/bin/top
...

Este arquivo também pode ser criado manualmente pelo administrador, utilizando um editor de textos ou um *script* que leia o conteúdo do arquivo `/etc/passwd` e os nomes dos binários presentes no sistema, gerando o arquivo automaticamente. Para fins de aprendizado e monitoração, está sendo utilizando o usuário que realmente executou comando que originou o processo (usuário real). O usuário efetivo do processo é ignorado, pois comandos com o *bit* `suid` ativado – o comando `su`, por exemplo – poderiam ser executados por usuários não autorizados, falseando a criação da base histórica e a monitoração.

Finalmente, o IDS gera uma relação seqüencial de usuários que utilizaram o sistema e os processos que foram executados durante o ciclo de vida da VM. Esta relação é utilizada na ACL para detectar anomalias com relação a usuários e processos autorizados.

5.3.2 Monitoração e Resposta

O processo de **monitoração** é responsável por detectar desvios de comportamento dos processos na máquina virtual e tomar decisões em relação ao desvio encontrado. O IDS recebe as seqüências de chamadas de sistema e verifica a existência na base histórica, caso a seqüência não esteja relacionada o processo é classificado como *suspeito (*figura 5.4).

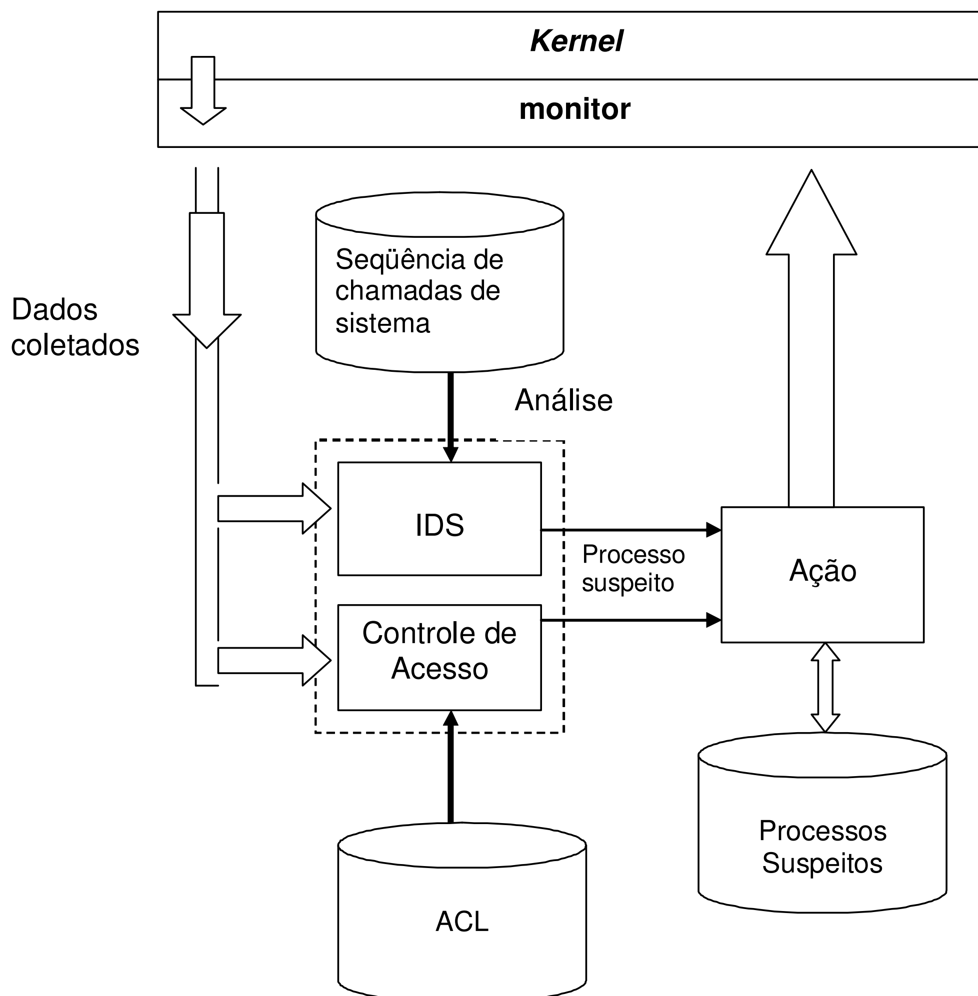

Figura 5.4 – IDS em modo Monitoração

Além de monitorar as chamadas de sistema efetuadas pelos processos do sistema convidado, o IDS também permite definir os usuários e processos autorizados (ACL) a utilizar a máquina virtual. Uma ACL define os direitos de acesso dos usuários e processos ao ambiente de execução virtual. Com isso é possível detectar invasões no sistema virtual ou impor limites aos usuários e processos legítimos da máquina virtual.

Processos suspeitos são restritos em seu acesso ao sistema operacional convidado, para evitar ações danosas. Assim, Todas as chamadas de sistema que podem ser utilizadas para se obter pleno acesso ao sistema operacional, classificadas com o nível de ameaça 1 [BER00, BER02] (tabela 5.4) e já descritas no capítulo 4, têm a sua execução negada para processos suspeitos. Dessa forma, o sistema operacional virtual pode isolar um processo suspeito sem causar impactos em outros processos que não dependam dele.

Tabela 5.4 – Chamadas de sistema negadas aos processos suspeitos

Tipo de Utilização	**Chamadas de Sistema**
Acesso ao sistema de arquivos e dispositivos	`open link unlink chmod lchown rename fchown chown mknod mount symlink fchmod`
Gerência de processos	`execve setgid setreuid setregid setgroups setfsuid setfsgid setresuid setresgid setuid`
Gerência de módulos	`init_module`

Se um processo for considerado suspeito, o IDS irá retornar 0 (falso) para o monitor, senão será retornado 1 (verdadeiro). Uma vez que um processo for considerado suspeito, todas as chamadas de sistema de nível 1 solicitadas por ele terão a sua execução negada. O controle de processos suspeitos é realizando dentro do IDS por uma tabela interna (memória) de identificadores de processos (*Process ID* – PID) versus nome do processo (*Process Name*). Não foi utilizando apenas o PID para o controle dos processos suspeitos para evitar que novos processos criados com o mesmo PID fossem afetados pelo IDS.

Utilizando o conjunto de seqüências de chamadas de sistema montadas durante a fase de aprendizado, o IDS detecta anomalias na seqüência das chamadas de sistema, utilizando o algoritmo proposto por [FOR96, HOF98], do processo e classifica o processo como suspeito.

5.3.3 Controle de Acesso

Utilizando a estrutura de ACLs – que são mecanismos que implementam o controle de acesso para os recursos de um sistema, através da enumeração das identidades dentro das entidades que compõem o sistema, permitindo o acesso para o recurso [SHI00] – é possível detectar o mau uso do sistema convidado. Uma lista de controle de acesso define os direitos de acesso dos usuários e processos ao ambiente de execução virtual. Com isso é possível detectar invasões no sistema virtual ou impor limites aos usuários e processos legítimos da máquina virtual.

A tabela 5.5 exemplifica o modelo de controle de acesso implementado no IDS, na tabela 5.5 o símbolo + indica a permissão, − indica a negação e `*` indica todos usuários e comandos. No exemplo, caso o usuário `alice` tentar utilizar o comando `ftp`, o processo por ela lançado será classificado como suspeito, pois este usuário não esta autorizado pela ACL a utilizar o comando. Da mesma forma, todos os processos lançados pelo usuário `charles`,

que não está na ACL e indicado pelo símbolo *, serão classificados como suspeitos. O mesmo ocorre com processos que não constem da ACL, mesmo que lançados por usuários válidos.

Tabela 5.5 – Exemplo de ACL

	ps	find	ssh	ftp	su	*
root	+	+	+	+	+	-
alice	+	+	-	-	-	-
bob	+	+	-	+	-	-
*	-	-	-	-	-	-

Durante o aprendizado, o IDS monta de forma automática uma lista seqüencial de usuários e processos. Esta lista (na realidade uma ACL otimizada para pesquisa) é utilizada para fins de autorização, ou seja, a lista indica que aquele usuário ou processo é válido dentro do sistema. Caso um usuário que não esteja nesta lista, lance um processo, tem todos os seus processos classificados como suspeitos. O mesmo ocorre com um processo lançado não esteja nesta lista, mesmo que lançado por um usuário válido do sistema.

5.3.4 Capacidade de reação

A reação a um provável ataque ou invasão é realizada através de alterações no código do monitor. Este pode ser alterado para, por exemplo, suspender a execução dos processos suspeitos ou impedir a execução de novos processos. Além da atuação sobre o sistema convidado, o sistema de detecção pode também interagir com o *firewall* que liga o sistema convidado à rede externa, bloqueando portas e conexões conforme necessário.

5.4 Benefícios e Limitações da Proposta

A arquitetura apresentada torna o sistema de detecção/resposta inacessível a eventuais invasores. Fazendo a analogia com um sistema convencional, é como se o sistema de detecção/resposta tivesse sido implementado no hardware da máquina, tornando-o incontornável aos processos dos usuários. A arquitetura traz outros benefícios para a segurança de sistemas:

- A arquitetura proposta é simples, de fácil compreensão e implementação;

- Pode ser implementado em outras VMs do tipo II;
- Não existe a necessidade de alterações no sistema anfitrião, pois serão criados processos distintos que irão executar sobre o sistema anfitrião;
- Independência do sistema anfitrião.

Como o IDS irá executar no sistema anfitrião, este tem algumas restrições e deficiências de segurança e desempenho.

Restrições:

- As interações com o sistema convidado sempre devem ser feitas através do monitor de máquina virtual;
- O monitor de máquina virtual deve ser inacessível aos processos de usuário do sistema convidado (propriedade de isolamento);
- Todos os serviços de rede devem ser providos pelos processos do sistema convidado. O acesso via rede ao sistema real subjacente deve ser evitado;
- O sistema real (anfitrião) deve ser considerado confiável (*trusted*) – Caso o sistema anfitrião seja atacado, toda e qualquer informação com origem neste sistema deve ser considerada suspeita. Se for considerado que somente pode haver acesso a esta máquina a partir do console (ou seja, somente com a presença do administrador), qualquer outro acesso deverá ocorrer através do sistema convidado e qualquer tentativa de ataque a partir do sistema convidado, será detectada em tempo hábil no sistema real e repelida.

Deficiências:

- Custo ocasionado pelo sistema de aprendizado e monitoração, acrescida do custo da virtualização;
- O código do monitor deve ser alterado para reconhecer todas as decisões tomadas pelo mecanismo de detecção de intrusão e providenciar as respectivas ações;
- O processo de criação de uma base histórica suficientemente confiável para minimizar a ocorrência de falso positivos/negativos durante a monitoração pode ser trabalhoso.

5.5 Trabalhos Correlatos

O artigo [CHE01] cita alguns benefícios que a alteração do código das máquinas virtuais podem trazer para a segurança e compatibilidade de sistemas, como a captura de mensagens de *log*, sistema de detecção de intrusão agindo sobre a máquina virtual (através do controle de estados) ou um ambiente de migração de sistemas. Todavia, o artigo citado não sugere como devem ser implementadas essas alterações. As próximas seções descrevem os principais projetos relacionadas a segurança utilizando detecção de intrusão, máquinas virtuais ou alterações no sistema anfitrião.

5.5.1 Projeto Revirt

Em [DUN02] é descrita uma implementação para garantir a segurança do sistema virtual, esta implementação prevê uma camada intermediária entre o monitor e o sistema anfitrião. Esta camada, chamada de *Revirt*, é responsável pela captura dos dados enviados através do `syslog` (*daemon* padrão Unix que registra as informações enviadas pelas aplicações em execução) da máquina virtual, os enviando para registro no sistema anfitrião.

Para a implementação da proposta foi utilizado o sistema *UMLinux* (máquina virtual) e o Linux (sistema anfitrião) com a versão do *kernel* 2.4.18, ambos com o código modificado. O monitor foi alterado, utilizando um módulo de *kernel*, para realizar o envio das informações coletadas para um *buffer* circular na memória do sistema anfitrião, o sistema anfitrião foi alterado para retirar as informações desta área de memória e armazenar em arquivo similar ao gerado pelo `syslog`.

Esta abordagem permite a análise posterior no sistema real através de métodos tradicionais de detecção de intrusão. Caso o sistema virtual seja atacado e subvertido, as mensagens coletadas podem estar sendo manipuladas pelo atacante e conseqüentemente não são mais confiáveis (pois essas informações são geradas pelo *daemon* `syslog` que executa no *user-space* do *kernel* convidado). Esta situação não está prevista na implementação, mas pode ser minimizada através da utilização na VM de mecanismos tradicionais de IDS.

5.5.2 Projeto VMI IDS

O trabalho que mais se aproxima da nossa proposta está descrito em [GAR03]. Esse trabalho propõe uma arquitetura para a detecção de intrusão em máquinas virtuais

denominada VMI IDS (*Virtual Machine Introspection Intrusion Detection System*). Sua abordagem considera o uso de um monitor de tipo I, executando diretamente sobre o hardware. O IDS executa em uma das máquinas virtuais e observa dados obtidos das demais máquinas virtuais, buscando evidências de intrusão. Somente o estado da máquina virtual é analisado, sem levar em conta as atividades dos processos nela contidos. O sistema age de forma passiva, pois sua capacidade de resposta é limitada: caso haja suspeita de intrusão, a máquina virtual invadida é suspensa até que essa suspeita seja confirmada (nesse caso, a mesma é reiniciada).

Essa abordagem difere de nossa proposta nos aspectos de detecção de intrusão e ação em caso de intrusão. Nossa proposta permite analisar processos isoladamente, detectando atividades anômalas e impedindo intrusões a partir dos mesmos. Isto permite que processos válidos de um sistema em produção não sejam afetados em sua operação. Além disso, não há necessidade de suspender a máquina virtual para confirmação da intrusão. Outra característica única de nossa proposta é o emprego de um modelo de autorização para usuários e processos, construído de forma semi-automática, baseado na utilização do sistema convidado durante a fase de aprendizado.

5.5.3 LIDS – Linux Intrusion Detection System

Os sistemas Unix e Linux tem algumas características que prejudicam a segurança do sistema operacional, tais como, sistemas de arquivos e processos que não são protegidos, o usuário `root` detém todo o poder sobre o sistema operacional, o método de autenticação não é seguro e modelo de controle de acesso é ineficiente [KLE03].

O LIDS é um *patch* para o *kernel* do sistema operacional Linux que implementa uma série de controles adicionais sobre os recursos do sistema. Entre estes controles estão:

- **Proteção de arquivos** – Os arquivos protegidos pelo LIDS não podem ser modificados por nenhum usuário, inclusive o usuário `root`; estes arquivos podem ser escondidos dos demais componentes do sistema operacional;
- **Proteção de Processos** – Os processos podem ser protegidos inclusive da ação do usuário `root`, e, assim como os arquivos, podem ser escondidos;
- **Implementação de ACLs** – O LIDS implementa o controle de recursos do sistema operacional através da utilização de ACLs de forma similar a nossa proposta;
- **Bloqueio de Recursos** – Bloqueio de acesso a vários recursos do sistema

operacional, como portas TCP/IP, acesso a dispositivos de *hardware*, entre outros.

Embora o LIDS não implemente nenhum controle de integridade de arquivos, ele possui características para a garantia da segurança do sistema operacional, sendo possível limitar até os poderes do usuário `root`. O LIDS não utiliza VMs para garantir a integridade do sistema anfitrião, mas a subversão da máquina só é possível através de ataques direcionados para o *kernel* do sistema.

5.5.4 Projeto Systrace

O projeto Systrace [PRO03], tem por finalidade autorizar a um processo não privilegiado a execução de algumas operações com um privilégio elevado. Esta autorização é configurada através de uma política que elimina a necessidade de binários com o `suid` *bit* ou `sgid` *bit* ativados. O Systrace suporta o confinamento de processos e permite a detecção de intrusão. A geração de políticas pode ocorrer de forma automatizada durante o acompanhamento da execução de um processo. As políticas descrevem o comportamento desejado dos serviços ou aplicações de usuário no sistema operacional. As políticas são gerenciadas por um mecanismo específico.

O controle e a criação das políticas são realizados através do controle de execução das chamadas de sistema e seus argumentos. No momento da execução de uma chamada de sistema de um processo que esteja sendo monitorado, é consultada a política construída para aquele processo e verificada a validade dos seus argumentos. Se a política atual não permite a execução da chamada de sistema, o usuário é consultado para autorizar ou negar a execução. Caso o usuário autorize a execução, esta nova instrução entra no mecanismo de políticas do Systrace para consultas posteriores. Se a política ou o usuário não permitirem a execução da instrução, o mecanismo de interceptação nega a execução da chamada de sistema.

Atualmente, existem versões do Systrace para os sistemas operacionais Linux, Mac OS X, FreeBSD, NetBSD e OpenBSD.

5.6 Conclusão

Este capítulo descreveu uma proposta para aumentar a segurança de sistemas computacionais através do uso de máquinas virtuais. A base da proposta é monitorar as ações dos processos da máquina virtual através de um sistema de detecção de intrusão externo à mesma. Os dados usados para a detecção de intrusão são obtidos por interação entre o monitor

de máquina virtual e o processo que implementa o IDS na máquina real subjacente. Com isso, o sistema de detecção torna-se inacessível aos processos da máquina virtual e não pode ser subvertido por um eventual invasor.

No próximo capítulo serão descritos detalhes do protótipo implementado e das experiências realizadas para validar seu funcionamento e avaliar seu desempenho.

Capítulo 6

Implementação e Resultados

Um protótipo da proposta, apresentado no capítulo anterior, foi implementado em plataforma Linux, usando o monitor de máquina virtual UML (*User-Mode Linux*) [DIK00], com *kernel* Linux 2.6.1. Na máquina real foi implementado um programa, atualmente denominado VMIDS (*Virtual Machine Intrusion Detection System*), responsável pelo registro e análise das informações enviadas pelo monitor. As próximas seções descrevem as alterações na VM, a implementação dos módulos e avaliam o funcionamento do sistema.

6.1 Alterações na Máquina Virtual

O monitor UML foi modificado para permitir extrair informações detalhadas sobre o sistema convidado, como por exemplo, o fluxo de chamadas de sistema de seus processos.

No UML, a virtualização das chamadas de sistema é implementada através de uma *thread* de rastreamento [DIK00] que intercepta e redireciona todas as chamadas de sistema dos processos convidados para o *kernel* virtual. A arquitetura do monitor UML executa as chamadas de sistema dos processos em um módulo chamado `execute_syscall`. Esse módulo foi alterado para realizar a chamada de outro módulo, chamado `registra_syscall`, que é o responsável pela comunicação com o sistema de detecção de intrusão em execução no sistema anfitrião (Figura 6.1).

A comunicação entre o monitor do UML e o processo IDS está sendo feita através de *named pipes*. Dessa forma, o próprio sistema operacional anfitrião gerencia o fluxo de informações entre esses processos. Esta forma de comunicação foi utilizada para simplificar o protótipo na sincronização da comunicação com o monitor, já que o sistema anfitrião é

otimizado para este tipo de comunicação.

O novo módulo (`registra_syscall`) envia as informações sobre cada processo para o IDS na máquina real toda vez que o processo executar uma chamada de sistema. São enviadas para o IDS: seu dono (*user id*), número do processo perante o sistema operacional (*process id*), nome do comando que gerou o processo (*process name*) e a chamada de sistema (sem parâmetros).

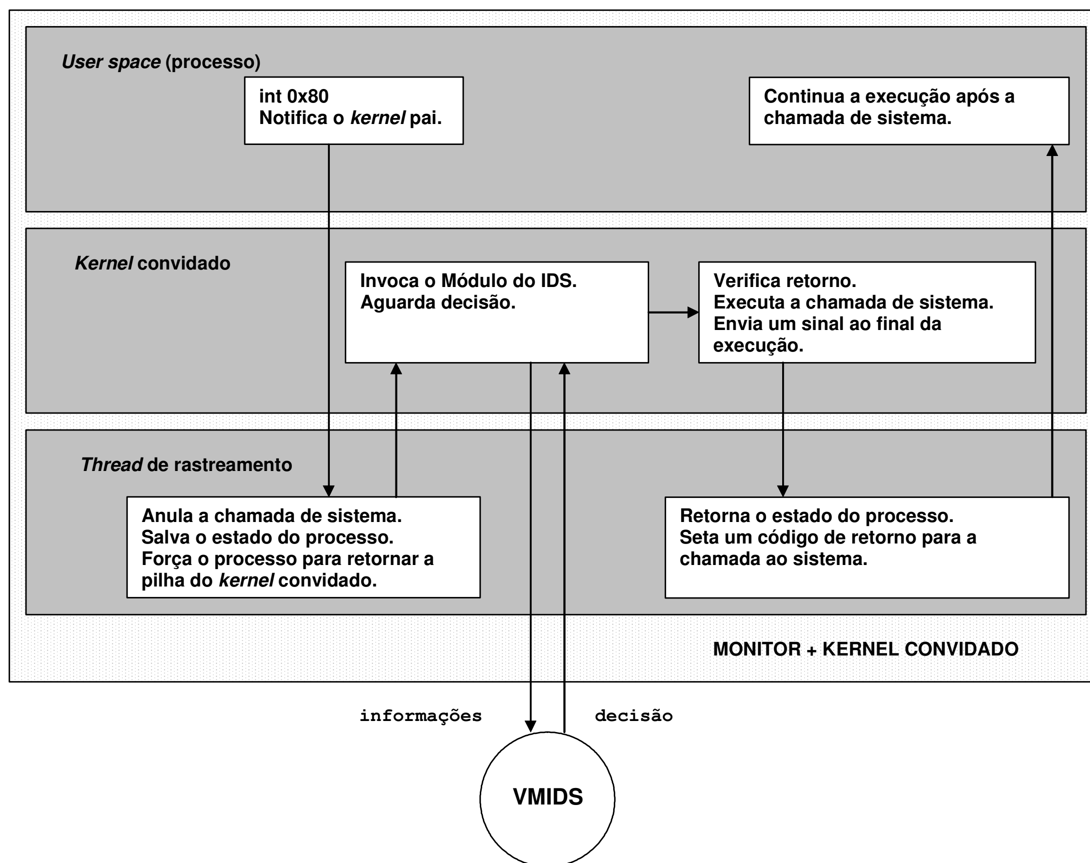

Figura 6.1 – Alterações no *Kernel* do Monitor

O monitor espera o retorno do módulo (`registra_syscall`), caso o retorno seja 1 (verdadeiro) o monitor continua com o seu código normal, ou seja, executa a chamada de sistema e devolve o controle para o processo. Caso o retorno seja 0 (falso), a chamada de sistema não é executada e é retornado o erro `-EFAULT` para a chamada de sistema, este valor indica uma falha na execução da chamada de sistema e normalmente os programas, que geraram os processos, já estão preparados para tratar os erros retornados após a chamada de

uma chamada de sistema. Por exemplo, se a chamada de sistema `open`, responsável por abertura de arquivos, retornar um código de falha de abertura, o programa ou tenta abrir novamente o arquivo (e recebe um novo código de falha) ou emite um aviso de erro e encerra o processo. O protótipo da implementação atual está ilustrado na figura 6.2.

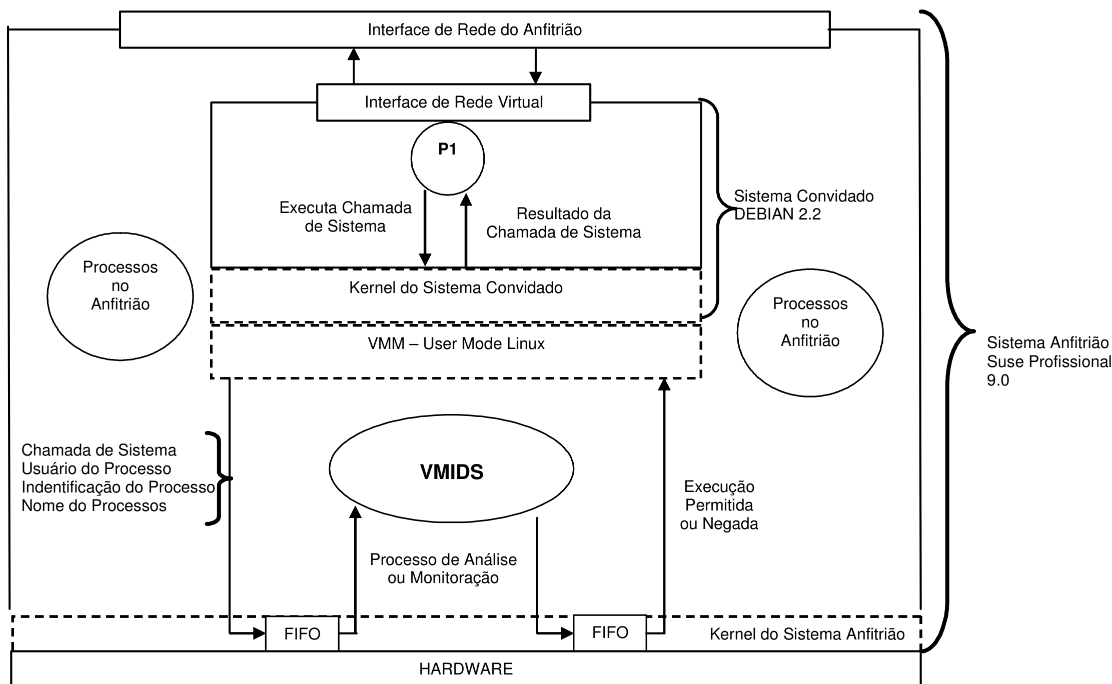

Figura 6.2 – Protótipo Implementado

6.2 Avaliação do Protótipo

O equipamento utilizado para avaliação do protótipo nos testes foi um Athlon XP 1600, com 512 MB de memória, 30 GB de disco rígido IDE. O sistema operacional anfitrião (*host*) utilizado foi o *Suse Linux Professional 9.0* com o *kernel* 2.4.21 otimizado para o processador AMD. Como sistema operacional convidado foi utilizado um imagem do sistema de arquivos do *Linux Debian 2.2* executando *kernel* 2.6.1 do UML. O sistema de arquivos utilizado e o *kernel* estão disponíveis na página do projeto UML [DIK00]. O UML foi escolhido para a implementação do protótipo por ter o seu código fonte disponível e por ser suportado pelo *kernel* oficial do Linux a partir da versão 2.6.

As próximas seções discutem os aspectos relacionados a custo, testes de detecção de intrusão e análise dos resultados obtidos.

6.2.1 Custo

Para avaliar o impacto da proposta sobre os processos mais comuns de um sistema Linux, foram efetuadas algumas medidas de tempo de execução dentro da máquina virtual. Para realizar as medidas, foi utilizado o resultado do comando `time` do sistema convidado.

Foram medidos os tempos de execução dos comandos `find`, `who`, `ls` e `ps` em quatro situações distintas: 1) na máquina real, 2) na máquina virtual sem modificações, 3) na máquina virtual com o VMIDS em modo aprendizado e 4) na máquina virtual com o VMIDS em modo monitoração. A utilização destes comandos representa os piores casos de utilização de um processo, devida a grande quantidade de chamadas de sistemas de E/S realizadas. Como estes processos executam sem interação com o usuário, as operações e E/S representam a maior parte de seu tempo de processamento. Os parâmetros utilizados para compilação e execução das instâncias das máquinas virtuais foram os mesmos.

Tabela 6.1 – Tempo Médio de Execução (em milisegundos)

Comando	**Quantidade de chamadas de sistema**	**Número de seqüências (de tamanho 3)**	**Máquina Real**	**Máquina Virtual**		
				original	**aprendizado**	**monitoração**
`ps -ef`	925	134	22	40	67	70
`find / >/dev/null 2>&1`	6958	94	41	146	355	388
`ls -laR / >/dev/null 2>&1`	18096	121	166	371	797	819
`who`	224	61	2	13	33	16

A tabela 6.1 apresenta os tempos médios de execução de cada comando, cada medida de tempo foi efetuada 10 vezes, e os desvios observados foram sempre inferiores a 4% do valor médio. Pode-se observar que os tempos médios dos comandos, quando executados dentro do ambiente virtual, são muito superiores aos obtidos diretamente sobre a máquina real. Isso demonstra o custo computacional envolvido na virtualização, conforme discutido anteriormente, e as limitações do monitor UML cujo desempenho é inferior a monitores comerciais como o VMware.

Tabela 6.2 – Acréscimo de Tempo

Comando	Máq. virtual original em relação à máquina real	VMIDS em modo aprendizado em relação à máquina virtual original	VMIDS em modo monitoração em relação à máquina virtual original
`ps -ef`	76,79%	69,19%	77,78%
`find / >/dev/null 2>&1`	257,07%	142,62%	164,75%
`ls -laR / >/dev/null 2>&1`	124,00%	114,54%	120,62%
`who`	458,33%	149,25%	19,40%

A tabela 6.2 apresenta o custo ocasionado pelas alterações na máquina virtual para responder ao sistema de aprendizado, monitoração e resposta. A diferença entre os tempos apresentados pelos comandos `ps`, `who`, `find` e `ls` se deve à quantidade de chamadas de sistema que cada um realiza durante sua execução e ao custo de aprendizado ou monitoração, sendo que em alguns casos (comando `who`) o custo do aprendizado é maior que o custo da monitoração.

6.2.2 Rootkits Utilizados Para os Testes

Além dos testes de desempenho, foram realizados testes de detecção de intrusão através da utilização de alguns *rootkits* (tabela 6.3). Estes *rootkits* alteram vários comandos do sistema operacional original para evitar que sejam detectados (ocultando os processos ou arquivos do invasor) e para registrar as informações de usuários e senhas digitados (via modificações nos comandos `telnet`, `sshd` e `login`). Todos os *rootkits* utilizados estão disponíveis em *http://www.antiserver.it/Backdoor-Rootkit/*.

Tabela 6.3 – Rootkits utilizados para validar o VMIDS

Nome	***Descrição***
`Adore`	Oculta arquivos, diretórios, processos, fluxo de rede. Instala um *backdoor* e um programa de usuário para controlar tudo.
`ARK 1.0`	*Ambient's Rootkit for Linux*. Composto somente de binários. Inclui versões com *backdoor* dos comandos `syslogd`, `login`, `sshd`, `ls`, `du`, `ps`, `pstree`, `killall`, e

	netstat.
Knark v.2.4.3	Oculta arquivos, fluxo de rede, processos e redireciona a execução de programas.
hhp-trosniff	Conjunto completo de modificações do ssh, ssh2, sshd2 e openssh, para extrair e registrar a origem, destino, nome do *host*, nome do usuário e senha.
login.tgz	Pacote do login para Linux, mas com um *backdoor*.

6.2.3 Testes por Anomalia na Seqüência de Chamadas de Sistema

Após a criação da base histórica (VMIDS executando no modo aprendizado), foram realizados testes de detecção de intrusão utilizando os comandos alterados pelos *rootkits* citados anteriormente. Para os testes de detecção através de análise de seqüências de chamadas de sistema, adotou-se que o atacante já havia subvertido (através de um processo que não estava sendo monitorado) a máquina e instalou os *rootkits*, sendo detectada a intrusão somente após a primeira utilização do comando. Nos testes realizados, o VMIDS detectou todas as modificações causadas por estes *rootkits*.

Por exemplo, o apêndice B (tabela B.1) contém as seqüências de chamadas de sistema (não repetidas) válidas para o comando login, a tabela B.2 (apêndice B) contém as seqüências de chamadas de sistema (não repetidas) do comando login alterado.

O comando login foi alterado para liberar o acesso ao sistema com uma senha específica (mesmo que o usuário seja inválido), o acesso é liberado sempre com poderes de administrador (root) e todos os acessos válidos (usuário e senha) são registrados em um arquivo para utilização posterior.

A tabela B.1 contém 232 seqüências de chamadas de sistema, a tabela B.2 contém 229 seqüências. Embora o comando alterado realize um maior número de instruções (abrir arquivo, gravar informações e fechar arquivo), a diferença é ocasionada pelas combinações possíveis de chamadas de sistema para a geração do conjunto final.

A tabela 6.4 contém as seqüências de chamadas de sistema geradas pelo comando alterado e que não se encontram na base histórica confiável. Esta seqüência caracteriza a anormalidade no sistema.

Tabela 6.4 – Seqüências de chamadas de sistema inválidas

```
{_llseek,write,close}, {rt_sigaction,rt_sigprocmask,exit},
{rt_sigprocmask,exit,uname}, {fstat64,_llseek,write},
{fstat64,mmap,fstat64}, {write,close,munmap}, {mmap,fstat64,_llseek},
{mmap,read,uname}, {mmap,read,write}, {munmap,rt_sigaction,rt_sigaction},
{munmap,setpriority,open}, {setpriority,open,fstat64}
```

A análise de chamadas de sistema somente ocorre para os comandos definidos pelo administrador do sistema. O administrador deverá criar um arquivo com a relação dos comandos que devem ter suas chamadas de sistema registradas (na fase aprendizado) e monitoradas.

6.2.4 Testes com Utilização de ACLs

Para os testes com a utilização de ACLs, foram criados usuários no sistema virtual, mas não foi concedido nenhum privilégio para os mesmos. Caso algum usuário não autorizado tente utilizar um comando, o processo gerado por este comando tinha suas chamadas de sistema negadas (conforme processo descrito anteriormente).

Utilizando a tabela 6.5 como um exemplo da ACL que é criada durante a fase de aprendizado (pode ser criada manualmente também), o usuário `root` (UID 0) pode executar os comandos `login`, `bash`, `mesg`, `ls`, `ps`, `halt`, `shutdown` e `find`. O usuário `marcos` (UID 1001) possui autorização para executar os comandos `login`, `bash` e `ls`. Caso o usuário `marcos` utilizasse o comando `find`, o processo seria classificado como suspeito. O mesmo ocorre com o `root`, caso ele queira utilizar o comando `mount` (que não está na ACL).

Tabela 6.5 – ACL gerada pelo VMIDS durante a fase de aprendizado

```
0:/bin/login
0:/bin/bash
0:/usr/bin/mesg
0:/bin/ls
0:/bin/ps
0:/usr/bin/halt
0:/sbin/shutdown
0:/usr/bin/find
1001:/bin/login
1001:/bin/bash
1001:/bin/ls
```

Durante a fase de aprendizado, o VMIDS cria 2 listas de controle de acesso. A primeira lista contém a relação dos processos que podem ser executados e a segunda lista contém a relação dos usuários que podem utilizar o sistema. Estas listas foram criadas para os casos que não se deseja utilizar a ACL convencional para o controle de processos e usuários no sistema. Neste caso, todos os usuários constantes da lista (tabela 6.6) podem executar qualquer processo da lista de processos (tabela 6.7).

Tabela 6.6 – Lista de usuários autorizados

```
0
1001
```

Tabela 6.7 – Lista de processos autorizados

```
/bin/rm
/bin/cat
/bin/uname
/bin/hostname
/bin/sh
/bin/date
/bin/cp
/usr/bin/find
/bin/chmod
/bin/chgrp
/bin/sed
/bin/mv
/bin/login
/bin/bash
/usr/bin/mesg
/bin/ls
/bin/ps
/usr/bin/tty
/usr/bin/halt
/sbin/shutdown
```

Os usuários `root` (UID 0) e `marcos` (UID 1001) podem executar qualquer comando da tabela 6.7. Caso algum usuário, que não está na relação de usuários autorizados, utilizar o sistema o VMIDS irá classificar todas as suas ações (processos) como suspeitos. E se algum usuário autorizado utilizar um comando não autorizado (não constante na lista de processos autorizados), este processo será classificado como suspeito.

6.3 Análise dos Resultados Obtidos

Nos testes realizados ficou evidente a efetividade e complementaridade de ambos os

mecanismos implementados pelo VMIDS: a lista de controle de acesso impede a execução de binários desconhecidos, enquanto o mecanismo de IDS detecta e impede a execução de binários conhecidos, mas adulterados. O mecanismo de IDS por análise de chamadas de sistema também detecta ataques de *buffer overflow* ou *format string* (desde que estes ataques acabem gerando seqüências de chamadas de sistema inválidas), esta é uma característica dos algoritmos de detecção de intrusão por análise de chamadas de sistema.

Esta complementaridade fica evidente para os ataques que utilizam *exploits* para ganhar o acesso a uma máquina através de vulnerabilidades do software. Conforme descrito em [HON01], após um ataque, é normal o atacante criar uma conta de usuário comum e outra com poderes de super-usuário no sistema atacado. O atacante conecta-se a máquina através de um serviço válido (`telnet,` por exemplo), e utiliza o comando `su` para acessar a nova conta `root`, e posteriormente procede com a instalação de um *rootkit* para garantir a efetividade do ataque. Neste cenário, o VMIDS detecta o ataque ao serviço através da análise das chamadas de sistema, impedido que o *exploit* obtenha sucesso para ganhar o *shell* do usuário. Se o VMIDS não estiver monitorando o processo em questão, o atacante é detectado através do controle de ACLs, no momento em que for utilizar a nova conta de usuário, e impedirá a execução do comando `su` (através da negação das chamadas de sistema).

6.4 Considerações Finais

O protótipo construído demonstra que a abordagem é viável e apresenta um desempenho satisfatório (embora prejudicado devido a utilização de um monitor ainda em desenvolvimento e com desempenho inferior a outros monitores comerciais existentes). Todavia, trabalhos complementares devem ser realizados para melhorar o desempenho dos mecanismos de detecção de intrusão e respostas atualmente implementados.

O objetivo principal do projeto, impedir a continuidade da execução do processo suspeito na máquina virtual e conseqüentemente a possibilidade de corrupção do sistema, foi alcançado com o protótipo atual.

6.5 Conclusão

Este capítulo descreveu o protótipo do IDS e as alterações no VMM, conforme proposto no capítulo 5. A implementação demonstrou a eficácia dos mecanismos adotados para a detecção de intrusão e controle dos serviços (processos) disponíveis no ambiente.

Capítulo 7

Conclusão

Este trabalho descreve uma proposta para aumentar a segurança de sistemas computacionais através do uso de máquinas virtuais. A base da proposta é monitorar as ações dos processos da máquina virtual através de sistemas de detecção de intrusão externos à mesma. Os dados usados para a detecção de intrusão são obtidos por interação entre o monitor de máquina virtual e o processo que implementa o IDS na máquina real. Com isso, o sistema de detecção torna-se inacessível aos processos da máquina virtual e não pode ser subvertido por um eventual invasor.

O protótipo implementado (denominado VMIDS), embora seja funcional, ainda não está com um desempenho adequado para ser utilizado em sistemas de produção. Os algoritmos de registro e pesquisa de informações devem ser melhorados e otimizados. Atualmente estamos trabalhando no sentido de melhorar o desempenho da análise das informações – trabalhando com bancos de dados em vez de arquivos textos e registros em memória – e no controle dos processos, como sincronizar a tabela de controle de processos suspeitos com a tabela de processos encerrados da VM. Outra questão importante relacionada a desempenho é o uso do ambiente UML. Certamente o desempenho obtido com o protótipo seria melhor com o uso de um ambiente de máquinas virtuais comercial, mas não tempos acesso ao código-fonte dos mesmos para implementar a proposta.

Outros aspectos em estudo dizem respeito a implementar formas de monitoração baseadas em outras informações, como a análise de fluxo de rede da máquina virtual, a alocação de memória e o comportamento dos usuários sobre determinados processos. Algoritmos mais sofisticados de detecção de intrusão podem ser implementados a partir

dessas informações, auxiliando a reduzir a ocorrência de resultados falsos (positivos e negativos).

A principal contribuição deste trabalho foi propor e implementar uma arquitetura para a segurança de sistemas computacionais fazendo uso de ambientes de máquinas virtuais e de sistema de detecção de intrusão. Resultados parciais deste trabalho foram publicados também em [LAU03].

Referências Bibliográficas

[AGR99] AGREN, O. *Teaching Computer Concepts Using Virtual Machines*. SIGCSE Bulletin, 1999. Vol. 31, P. 84 – 85.

[ALL99] ALLEN, J. et al. *State of the Practice of Intrusion Detection Technologies*. Technical Report CMU/SEI-99-TR028. Disponível em http://www.sei.cmu.edu/publications/documents/99.reports/99tr028/99tr028abstract.html. Carnegie Mellon University. Acessado em: 15/01/2003.

[AND95] ANDERSON, D. et al. (SRI International). *Detecting Unusual Program Behavior Using the Statistical Component of the Next-Generation Intrusion Detection Expert System (NIDES)*. (SRI-CSL-95-06). Menlo Park, CA: Computer Science Laboratory, SRI International, 1995. http://www.sdl.sri.com/nides/index5.html. Acessado em: 15/12/2003.

[ATT73] ATTANASIO, C. *Virtual Machines and Data Security*. Proceedings of the workshop on virtual computer systems, 1973. Cambridge, Massachusetts – USA. P. 206 – 209.

[BAR03a] BARHAM, P. et al. *Xen and the Art of Virtualization*. 19th ACM Symposium on Operating Systems Principles – SOSP 2003. P. 164-177.

[BAR03b] BARHAM, P. et al. *Xen 2002*. Technical Report Number 553, UCAM-CL-TR-553, ISSN 1476-2986. University of Cambridge, 2003.

[BEL73] BELPAIRE, A. e HSU, Nai-Ting. *Formal Properties of Recursive Virtual Machine Architectures*. Proceedings of the fifth ACM symposium on Operating systems principles, 1973. Austin – Texas – USA. P. 89 – 96.

[BER00] BERNASCHI, M., GRABRIELLI, E. e MANCINI, L. *Operating System Enhancements to Prevent the Misuse of System Calls*, Proceedings of the ACM Conference on Computer and Communications Security, 2000. P. 174 – 183.

[BER02] BERNASCHI, M., GRABRIELLI, E. e MANCINI, L. *REMUS: A Security-Enhaced Operating System*. ACM Transactions on Information and System Security, 2000. Vol. 5, Nº 01, P 36 – 61.

[BLU02] BLUNDEN, B. *Virtual Machine Design and Implementation in C/C++*. Wordware Publishing, 2002. Plano, Texas – USA.

[BSI02] British Standards Institute. *BS 7799-2 – Code of Practice for Information Security Management – Part 2: Specification for Information Security Management Systems*. Londres – UK, 2002.

[CER02] CERT/CC. *Vulnerability Note VU#726187 - HP-UX kernel specifies incorrect arguments for setrlimit()*, 2002. Disponível em: http://www.kb.cert.org/vuls/id/726187. Acessado em: 15/01/2004.

[CER03a] CERT/CC. *Advisory CA-2003-07 Remote Buffer Overflow in Sendmail*, 2003. Disponível em: http://www.cert.org/advisories/CA-2003-07.html. Acessado em: 29/02/2004.

[CER03b] CERT/CC. *Advisory CA-2003-25 Buffer Overflow in Sendmail*, 2003. Disponível em: http://www.cert.org/advisories/CA-2003-25.html. Acessado em: 29/02/2004

[CER03c] CERT/CC. *Advisory CA-2003-16 "Buffer Overflow" no Microsoft RPC*, 2003. Disponível em: http://www.nbso.nic.br/certcc/advisories/CA-2003-16-br.html. Acessado em:15/02/2004

[CHE01] CHEN, P. e NOBLE, B. *When Virtual Is Better Than Real*. Proceedings of the 2001 Workshop on Hot Topics in Operating Systems (HotOS), 2001.

[COW00] COWAN, C. et all. *Buffer overflows: attacks and defenses for the vulnerability of the decade*. In DARPA Information Survivability Conference and Exposition – DISCEX '00. Proceedings, 2000. Vol. 2, P. 119 – 129.

[DEN87] DENNING, D. *An Intrusion-Detection Model*. IEEE Transactions on Software Engineering, 1987. Vol.SE – 13, Nº 02, P. 222 – 232.

[DIK00] DIKE, J. *A User-mode port of the Linux Kernel*. Proceedings of the 4ª Annual Linux Showcase & Conference, 2000. Atlanta – USA. Disponível em http://user-mode-linux.sourceforge.net/als2000/index.html. Acessado em: 20/03/2003.

[DUN02] DUNLAP, G. et al. *ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay*. Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI), 2002. P. 211-224.

[ESK01] ESKIN, E., LEE, W. e STOLFO, S. *Modeling System Calls for Intrusion Detection with Dynamic Window Sizes*. DARPA Information Survivability Conference & Exposition II, 2001. Vol. 1, P. 165 – 175.

[FOR96] FORREST, S., HOFMEYR, S. e SOMAYAJI, A. *A sense of self for unix processes*, Proceedings IEEE Symposium on Research in Security and Privacy, 1996. P. 120 – 128.

[FRA03] FRASER, K. et al. *The Xenoserver Computing Infrastructure*. Technical Report Number 552, UCAM-CL-TR-552, ISSN 1476-2986. University of Cambridge, 2003.

[GAR03] GARFINKEL, T. e ROSENBLUM, M. *A Virtual Machine Introspection Based Architecture for Intrusion Detection*, Proceedings of the 2003 Network and Distributed System Security Symposium (NDSS), 2003.

[GHO99] GHOSH, A. e SCHWARTZBARD, A. *A Study in Using Neural Networks for Anomaly and Misuse Detection*. Proceedings of the Eight USENIX Security Symposium,

1999. P. 141 – 152.

[GOL73] GOLDBERG, R. *Architecture of Virtual Machines*. AFIPS National Computer Conference, 1973. New York – NY – USA.

[GOL74] GOLDBERG, R. *Survey of Virtual Machine Research*. IEEE Computer Magazine, 1974. Vol. 7, P. 34 – 45.

[GOL79] GOLDBERG, R. e MAGER, P. *Virtual Machine Technology: A bridge from Large Mainframes to Networks of Small Computers*. IEEE Proceedings Compcon Fall 79, 1979. P. 210 – 213.

[HEL97] HELMAN, P. e BHANGOO, J. *A Statistically base system for Prioritizing Information Exploration Under Uncertainty*. IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, 1997. Vol. 27, P. 449 – 466.

[HOF98] HOFMEYR, S., FORREST, S. e SOMAYAJI, A. *Intrusion Detection using Sequences of System Calls*, Journal of Computer Security,1998. Vol. 6, P. 151 – 180.

[HON01] HONEYNET PROJECT. *Know Your Enemy: Revealing the Security Tools, Tactics, and Motives of the Blackhat Community*. Addison-Wesley, 2001.

[HON03] HONEYNET PROJECT. *Know Your Enemy: Defining Virtual Honeynets – Differents types of Virtual Honeynets*. Disponível em http://project.honeynet.org/papers/virtual/, 2003. Acessado em: 15/12/2003.

[IBM99] IBM – Emergency Response Service. *IBM AIX Vulnerability in ptrace() system call*, 1999. Disponível em: http://ciac.llnl.gov/ciac/bulletins/j-055.shtml. Acessado em: 15/01/2004.

[ILG93] ILGUN, K. *USTAT: A Real-time Intrusion Detection System for UNIX*. Proceedings of the IEEE Symposium on Research on Security and Privacy, 1993. P. 16 – 28.

[INT98] Intel Corporation, Santa Clara, CA. *Intel Architecture.Developer's Manual*. Volumes I, II and III, 1998.

[KEL91] KELEM, N. e FEIERTAG, R. *A Separation Model for Virtual Machine Monitors*. Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on 1991, Oakland, California – USA. P. 78 – 86.

[KIN02] KING, S. e CHEN, P. *Operating System Extensions to Support Host Based Virtual Machines*. Technical Report CSE-TR-465-02, University of Michigan, 2002.

[KIN03] KING, S., DUNLAP, G. e CHEN, P. *Operating System Support for Virtual Machines*. Proceedings of the 2003 USENIX Technical Conference, 2003. P. 71 – 84.

[KLE03] KLEIN, S. *Linux Intrusion Detection System FAQ*. Disponível em http://www.lids.org/lids-faq/lids-faq.html, 2003. Acessado em: 15/12/2003.

[KOZ03] KOZIOL, Jack. *Intrusion Detection with Snor*t. Editora Sams – USA, 2003.

[LAU00] LAU, F. et al. *Distributed denial of service attacks*. In IEEE International Conference on Systems, Man, and Cybernetics, 2000. Vol. 3, P. 2275 – 2280.

[LAU03] LAUREANO, M., MAZIERO, C. e JAMHOUR, E. *Detecção de Intrusão em Máquinas Virtuais*. 5º Simpósio de Segurança em Informática – SSI. São José dos Campos, 2003. P. 1 – 7.

[LEE97] LEE, W., STOLFO, S. e CHAN, P. *Learning Patterns from Unix Process Execution Traces for Intrusion Detection*. In Proceedings of the AAAI-97 Workshop on AI Approaches to Fraud Detection and Risk Management, 1997. P. 50 – 56.

[LEE98] LEE, W. e STOLFO, S. *Data Mining Approaches for Intrusion Detection*. In Proceedings of the Seventh USENIX Security Symposium, 1998. P. 79 – 94.

[LIN99] LINDHOLM, T. e YELLIN, F. *Java Virtual Machine Specification – Second Edition*. Addison Wesley - 1999.

[MAL73] MALLACH, E. *On the Relationship Between Virtual Machines and Emulators*. Proceedings of the Workshop on Virtual Computer Systems, 1973. Cambridge – Massachusetts – USA. P. 117 – 126.

[NAC97] NACHENBERG, C. *Computer virus-antivirus coevolution*. Communications of the ACM archive, 1997. Vol. 40, P. 46 – 51.

[OZD94] ÖZDEN, B., GOLDBERG, A. e SILBERSCHATZ, A. *Virtual Computers – A New Paradigm for Distributed Operating Systems*. AT&T Bell Laboratories – Murray Hill – New Jersey – USA, 1994.

[PAX98] PAXSON, V. *Bro: A System for Detecting Network Intruders in Real-Time.* Proceedings of 7th USENIX Security Symposium. 1998. P. 31 – 52.

[POP74] POPEK, G. e GOLDBERG, R. *Formal Requirements for Virtualizable Third Generation Architectures*. Communications of the ACM, 1974. Vol 17, Nº 7, P 412 – 421.

[POR92] PORRAS, P. *STAT - A state transition analysis tool for intrusion detection*. M.S. thesis, Computer Science Dep., University of California Santa Barbara, 1992.

[POR96] PORRAS, P. e NEUMANN, P. *EMERALD: Conceptual Overview Statement*, http://www.sdl.sri.com/papers/emerald-position1/, 1996. Acessado em: 15/12/2003.

[POR97] PORRAS, P. e NEUMANN, P. *EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances*. http://www.sdl.sri.com/papers/emerald-niss97/, 1997. Acessado em 15/12/2003.

[PRO03] PROVOS, N. *Improving Host Security with System Call Policies*. Proceedings of the

12th USENIX Security Symposium. Washington – USA, 2003. P. 257 – 272.

[ROE99] ROESCH, M. *Snort - Lightweight Intrusion Detection for Networks*. Proceedings of the 13th Conference on Systems Administration, 1999. P 229 – 238.

[SAF00] Safer - Secury Alert for Enterprise Resources. *FreeBSD procfs Denial of Service Vulnerabilit*y, 2000. Disponível em: http://www.safermag.com/html/safer32/dos/05.html. Acessado em: 15/01/2004.

[SAF01] Safer – Secury Alert for Enterprise Resources. *Microsoft Windows 2000 Telnet System Call DoS Vulnerability*, 2001. Disponível em: http://www.safermag.com/html/safer38/dos/09.html. Acessado em: 15/01/2004.

[SCH97] SCHUBA, C. et al. *Analysis of a denial of service attack on TCP*. In Proceedings IEEE Symposium on Security and Privacy, 1997. P. 208 – 223.

[SEC03a] Secunia Stay Secure. *OpenBSD "semget()" Denial of Service Vulnerability*, 2003. Disponível em: http://www.secunia.com/advisories/9581/. Acessado em: 15/01/2004.

[SEC03b] Secunia Stay Secure. *Sun Solaris namefs Mounted Pipe and STREAMS Routines Denial of Service*, 2003. Disponível em: http://www.secunia.com/advisories/10007/. Acessado em: 15/01/2004.

[SEC03c] Secunia Stay Secure. *IBM AIX "getipnodebyname()" Denial of Service Vulnerability*, 2003. Disponível em: http://www.secunia.com/advisories/9901/. Acessado em: 15/01/2004.

[SHI00] SHIREY, R. “RFC2828 - Internet Security Glossary”, 2000.

[SIL00a] SILBERCHATZ, A. e GALVIN, P. *Sistemas Operacionais: Conceitos*. Prentice Hall, 2000. São Paulo – SP.

[SIL00b] SILBERSCHATZ, A.; GALVIN, P. e GAGNE, Greg. *Sistemas Operacionais: Conceitos e Aplicações*. Campus, 2000. Rio de Janeiro – RJ.

[SIR99] SIRER, E. et al. *Design and Implementation of a Distributed Virtual Machine for Network Computers*. Proceedings of the seventeenth ACM symposium on Operating systems principles, 1999. Charleston – Carolina do Sul – USA. P. 202 – 216.

[SOM03] SOMMER, R. e PAXSON, V. *Enhancing byte-level network intrusion detection signatures with context*. Proceedings of the 10th ACM conference on Computer and communication security, 2003. P 262 – 271.

[STA98] STALLINGS, W. *Cryptography and Network Security: Principles and Practice*. Prentice-Hall, 1998 – 2ª Edição. Upper Saddle River – New Jersey – USA.

[STA03] Stake, Inc. *MacOS X DirectoryService Privilege Escalation and DoS Attack*, 2003. Disponível em: http://www.atstake.com/research/advisories/2003/a041003-1.txt. Acessado em: 15/01/2004.

[STA04] STARZETZ, P. *Linux kernel do_mremap local privilege escalation vulnerability*, 2004. Disponível em: http://isec.pl/vulnerabilities/isec-0013-mremap.txt. Acessado em: 15/01/2004.

[SUG01] SUGERMAN, J., GANESH, V. e BENG-Hong Lim. *Virtualizing I/O Devices on Vmware Workstation's Hosted Virtual Machine Monitor*. Proceedings of the 2001 USENIX Annual Technical Conference. 2001. P. 1 – 14.

[TAN03] TANENBAUM, A. *Sistemas Operacionais Modernos*. Pretince Hall, 2003 – 2ª Edição. São Paulo – SP.

[VAR89] VARIAN, M. *VM and the VM Community: Past, Present, and Future*. Sessions of SHARE. Sessions 9059-9061. Melbourne – Australia, 1989 (última revisão agosto de 1997).

[VIG98] VIGNA, G. e KEMMERER, R. *NetSTAT: A Network-Based Intrusion Detection Approach.* Proceedings of the 14th Annual Computer Security Applications Conference, 1998. P. 25 – 35.

[VM99] VMware Inc. VMware Technical White Paper. Palo Alto – CA – USA, 1999.

[WAD00] WADLOW, T. *Segurança de Redes – Projeto e Gerenciamento de Redes Seguras*. Campus, Rio de Janeiro – RJ, 2000.

[WAR99] WARRENDER, C.; FORREST, S. e PERALMUTTER, B. *Detecting intrusions using system calls: alternative data models*. Proceedings IEEE Symposium Security and Privacy, 1999. P. 133 – 145.

[WHI02] WHITAKER, A. SHAW, M. e GRIBBLE, S. *Denali: A Scalable Isolation Kernel*. Proceedings of the Tenth ACM SIGOPS European Workshop, Saint-Emilion – França, 2002.

[YE00] YE, N. *A Markov Chain Model of Temporal Behavior for Anomaly Detection*. In Proceedings of the 2000 IEEE Systems, Man, and Cybernetics Information Assurance and Security Workshop, 2000. P. 171 – 174.

[ZON01] ZOVI, D. *Kernel Rootkits*. Sans Institute, 2001. Disponível em: http://www.sans.org/rr/papers/60/449.pdf. Acessado em: 14/01/2004.

Apêndice A

Exemplo de Seqüência de Chamadas de Sistema

Como apresentado no capítulo 5, a tabela de seqüências de chamadas de sistema foi elaborada a partir do resultado do comando `strace` monitorando o comando `who` (versão 5, de março de 2003 e utilizando a `glibc` versão 2.3.2), ambos do sistema Linux 2.4.21. O comando `strace` lista as chamadas de sistema executadas por um processo e seus parâmetros, enquanto o comando `who` lista os usuários conectados ao sistema operacional.

Como resultado do comando `who` foram obtidas as informações listadas:

```
root      tty1         Feb 16 18:05
marcos    :0           Feb 16 14:51 (console)
marcos    pts/0        Feb 16 14:51
marcos    pts/1        Feb 16 14:55
marcos    pts/2        Feb 16 14:56
marcos    pts/4        Feb 16 15:15
marcos    pts/6        Feb 16 18:04
marcos    pts/5        Feb 16 18:04
```

Como resultado do comando `strace` foram obtidas as informações listadas:

```
execve("/usr/bin/who", ["who"], [/* 63 vars */]) = 0
uname({sys="Linux", node="linux", ...}) = 0
brk(0)                                  = 0x804d7ec
old_mmap(NULL,  4096,  PROT_READ|PROT_WRITE,  MAP_PRIVATE|MAP_ANONYMOUS,  -1,
0) = 0x40019000
open("/etc/ld.so.preload",  O_RDONLY)       =  -1  ENOENT  (No  such  file  or
directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=58690, ...}) = 0
old_mmap(NULL, 58690, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4001a000
close(3)                                = 0
open("/lib/i686/libc.so.6", O_RDONLY)   = 3
read(3,  "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\320]\1"...,  512)  =
```

```
512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1461208, ...}) = 0
old_mmap(NULL, 1256644, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0x40029000
old_mmap(0x40155000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3,
0x12c000) = 0x40155000
old_mmap(0x4015a000, 7364, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x4015a000
close(3)                                = 0
munmap(0x4001a000, 58690)               = 0
open("/usr/lib/locale/locale-archive", O_RDONLY|O_LARGEFILE) = -1 ENOENT
(No such file or directory)
brk(0)                                  = 0x804d7ec
brk(0x806e7ec)                          = 0x806e7ec
brk(0)                                  = 0x806e7ec
brk(0x806f000)                          = 0x806f000
open("/usr/share/locale/locale.alias", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=2601, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x4001a000
read(3, "# Locale name alias data base.\n#"..., 4096) = 2601
read(3, "", 4096)                       = 0
close(3)                                = 0
munmap(0x4001a000, 4096)                = 0
open("/usr/lib/locale/pt_BR/LC_IDENTIFICATION", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=346, ...}) = 0
mmap2(NULL, 346, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4001a000
close(3)                                = 0
open("/usr/lib/locale/pt_BR/LC_MEASUREMENT", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=28, ...}) = 0
mmap2(NULL, 28, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4001b000
close(3)                                = 0
open("/usr/lib/locale/pt_BR/LC_TELEPHONE", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=54, ...}) = 0
mmap2(NULL, 54, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4001c000
close(3)                                = 0
open("/usr/lib/locale/pt_BR/LC_ADDRESS", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=132, ...}) = 0
mmap2(NULL, 132, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4001d000
close(3)                                = 0
open("/usr/lib/locale/pt_BR/LC_NAME", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=67, ...}) = 0
mmap2(NULL, 67, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4001e000
close(3)                                = 0
open("/usr/lib/locale/pt_BR/LC_PAPER", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=39, ...}) = 0
mmap2(NULL, 39, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4001f000
close(3)                                = 0
open("/usr/lib/locale/pt_BR/LC_MESSAGES", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=80, ...}) = 0
close(3)                                = 0
open("/usr/lib/locale/pt_BR/LC_MESSAGES/SYS_LC_MESSAGES", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=59, ...}) = 0
mmap2(NULL, 59, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40020000
close(3)                                = 0
open("/usr/lib/locale/pt_BR/LC_MONETARY", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=295, ...}) = 0
mmap2(NULL, 295, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40021000
close(3)                                = 0
```

```
open("/usr/lib/locale/pt_BR/LC_COLLATE", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=21499, ...}) = 0
mmap2(NULL, 21499, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40022000
close(3)                                = 0
open("/usr/lib/locale/pt_BR/LC_TIME", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=2356, ...}) = 0
mmap2(NULL, 2356, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40028000
close(3)                                = 0
open("/usr/lib/locale/pt_BR/LC_NUMERIC", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=59, ...}) = 0
mmap2(NULL, 59, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4015c000
close(3)                                = 0
open("/usr/lib/locale/pt_BR/LC_CTYPE", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=178468, ...}) = 0
mmap2(NULL, 178468, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4015d000
close(3)                                = 0
access("/var/run/utmpx", F_OK)              = -1 ENOENT (No such file or
directory)
open("/var/run/utmp", O_RDWR)           = -1 EACCES (Permission denied)
open("/var/run/utmp", O_RDONLY)         = 3
fcntl64(3, F_GETFD)                     = 0
fcntl64(3, F_SETFD, FD_CLOEXEC)         = 0
_llseek(3, 0, [0], SEEK_SET)            = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3, "\10\0\0\0\20\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 384) =
384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3, "\1\0\0\0005N\0\0~\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 384)
= 384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3, "\10\0\0\0\222\2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 384)
= 384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3, "\7\0\0\0r\n\0\0tty1\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 384)
```

```
= 384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3,  "\6\0\0\0s\n\0\0tty2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"...,  384)
= 384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3,  "\6\0\0\0t\n\0\0tty3\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"...,  384)
= 384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3,  "\6\0\0\0u\n\0\0tty4\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"...,  384)
= 384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3,  "\6\0\0\0v\n\0\0tty5\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"...,  384)
= 384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3,  "\6\0\0\0w\n\0\0tty6\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"...,  384)
= 384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
```

```
read(3, "\7\0\1@x\n\0\0:0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 384) =
384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3,  "\7\0\0\0\327\n\0\0pts/0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"...,  384)
= 384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3,  "\7\0\0\0\6\v\0\0pts/1\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"...,  384)
= 384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3, "\7\0\0\0%\v\0\0pts/2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 384) =
384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3, "\10\0\0\0iZ\0\0pts/3\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 384) =
384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3, "\7\0\0\0\17\2\0\0pts/4\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 384) =
384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
```

```
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3, "\7\0\0\0pE\0\0pts/6\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 384)
= 384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3, "\7\0\0\0oE\0\0pts/5\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 384)
= 384
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
alarm(0)                                = 0
rt_sigaction(SIGALRM, {0x40132da0, [], SA_RESTORER, 0x40052aa8}, {SIG_DFL},
8) = 0
alarm(1)                                = 0
fcntl64(3, F_SETLKW, {type=F_RDLCK, whence=SEEK_SET, start=0, len=0}) = 0
read(3, "", 384)                        = 0
fcntl64(3, F_SETLKW, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
alarm(0)                                = 1
rt_sigaction(SIGALRM, {SIG_DFL}, NULL, 8) = 0
close(3)                                = 0
stat64("/dev/tty1", {st_mode=S_IFCHR|0620, st_rdev=makedev(4, 1), ...}) = 0
time([1076965578])                      = 1076965578
open("/etc/localtime", O_RDONLY)        = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=721, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x40189000
read(3, "TZif\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\3\0\0\0\3\0"..., 4096)
= 721
close(3)                                = 0
munmap(0x40189000, 4096)                = 0
fstat64(1, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x40189000
stat64("/dev/:0", {st_mode=S_IFCHR|0622, st_rdev=makedev(4, 0), ...}) = 0
open("/usr/share/locale/pt_BR/LC_MESSAGES/coreutils.mo", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=58577, ...}) = 0
mmap2(NULL, 58577, PROT_READ, MAP_PRIVATE, 3, 0) = 0x4018a000
close(3)                                = 0
open("/usr/lib/gconv/gconv-modules.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=21436, ...}) = 0
mmap2(NULL, 21436, PROT_READ, MAP_SHARED, 3, 0) = 0x40199000
close(3)                                = 0
stat64("/dev/pts/0", {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...})
= 0
stat64("/dev/pts/1", {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 1), ...})
= 0
stat64("/dev/pts/2", {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 2), ...})
= 0
stat64("/dev/pts/4", {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 4), ...})
= 0
stat64("/dev/pts/6", 0xbffff280)        = -1 ENOENT (No such file or
directory)
stat64("/dev/pts/5", 0xbffff280)        = -1 ENOENT (No such file or
```

```
directory)
write(1, "root     tty1         Feb 16 18:"..., 353) = 353
close(1)                                = 0
munmap(0x40189000, 4096)                = 0
exit_group(0)                           = ?
```

Apêndice B

Seqüência de Chamadas de Sistema do Comando `login`

Este apêndice contém as seqüências de chamadas de sistema para o comando login, que é utilizado para demonstrar os testes de detecção de intrusão por análise de chamadas de sistema.

Tabela B.1 – Seqüências de chamadas de sistema do comando `login` original.

```
{socketcall,socketcall,rt_sigaction}, {socketcall,socketcall,close},
{socketcall,rt_sigaction,rt_sigprocmask}, {socketcall,rt_sigaction,write},
{socketcall,rt_sigaction,open}, {socketcall,rt_sigaction,munmap},
{socketcall,fcntl64,socketcall}, {socketcall,close,open},
{execve,wait4,rt_sigaction}, {vhangup,rt_sigaction,setsid},
{execve,uname,brk}, {exit,uname,brk}, {wait4,rt_sigaction,rt_sigaction},
{execve,brk,time}, {chdir,execve,uname}, {chdir,execve,brk},
{uname,ugetrlimit,close}, {uname,brk,open}, {uname,open,fcntl64},
{mprotect,mmap,close}, {mprotect,mmap,mmap},
{time,rt_sigaction,socketcall}, {time,write,close}, {time,open,fstat64},
{_llseek,time,write}, {_llseek,alarm,rt_sigaction}, {_llseek,read,read},
{_llseek,read,write}, {_llseek,read,open}, {_llseek,read,close},
{_llseek,write,fcntl64}, {chmod,chmod,setgid32}, {chmod,fstat64,chown32},
{chmod,setgid32,time}, {chmod,setgid32,brk}, {chmod,ioctl,rt_sigaction},
{nanosleep,write,write}, {rt_sigaction,socketcall,rt_sigaction},
{rt_sigaction,socketcall,fcntl64}, {rt_sigaction,vhangup,rt_sigaction},
{rt_sigaction,rt_sigaction,rt_sigaction},
{rt_sigaction,rt_sigaction,rt_sigprocmask},
```

```
{rt_sigaction,rt_sigaction,setuid32}, {rt_sigaction,rt_sigaction,brk},
{rt_sigaction,rt_sigprocmask,execve},
{rt_sigaction,rt_sigprocmask,nanosleep},
{rt_sigaction,rt_sigprocmask,rt_sigaction},
{rt_sigaction,rt_sigprocmask,fork}, {rt_sigaction,rt_sigprocmask,munmap},
{rt_sigaction,setuid32,chdir}, {rt_sigaction,alarm,rt_sigaction},
{rt_sigaction,alarm,fcntl64}, {rt_sigaction,alarm,alarm},
{rt_sigaction,alarm,close}, {rt_sigaction,alarm,gettimeofday},
{rt_sigaction,read,write}, {rt_sigaction,write,write},
{rt_sigaction,brk,brk}, {rt_sigaction,open,rt_sigaction},
{rt_sigaction,close,stat64}, {rt_sigaction,setsid,open},
{rt_sigaction,munmap,exit}, {rt_sigprocmask,execve,wait4},
{rt_sigprocmask,nanosleep,write},
{rt_sigprocmask,rt_sigaction,rt_sigprocmask},
{rt_sigprocmask,fork,rt_sigaction}, {rt_sigprocmask,munmap,exit},
{ugetrlimit,close,close}, {stat64,rt_sigaction,rt_sigaction},
{fstat64,chown32,chown32}, {fstat64,ioctl,mmap}, {fstat64,mmap,mprotect},
{fstat64,mmap,_llseek}, {fstat64,mmap,read}, {fstat64,mmap,close},
{getuid32,getegid32,setregid32}, {getegid32,setregid32,setreuid32},
{setreuid32,setregid32,getpid}, {setreuid32,access,setuid32},
{getpid,access,open}, {setregid32,getpid,access},
{setregid32,setreuid32,access}, {setgroups32,open,open},
{chown32,chmod,chmod}, {chown32,chmod,fstat64}, {chown32,chmod,ioctl},
{chown32,chown32,chmod}, {setuid32,chdir,execve},
{setuid32,setreuid32,setregid32}, {setgid32,time,rt_sigaction},
{setgid32,brk,time}, {fork,rt_sigaction,rt_sigaction},
{fcntl64,socketcall,socketcall}, {fcntl64,_llseek,alarm},
{fcntl64,_llseek,write}, {fcntl64,rt_sigaction,alarm},
{fcntl64,fstat64,mmap}, {fcntl64,fcntl64,_llseek},
{fcntl64,fcntl64,fstat64}, {fcntl64,fcntl64,close}, {fcntl64,read,fcntl64},
{fcntl64,close,close}, {alarm,rt_sigaction,rt_sigaction},
{alarm,rt_sigaction,alarm}, {alarm,getuid32,getegid32},
{alarm,fcntl64,_llseek}, {alarm,fcntl64,read}, {alarm,alarm,rt_sigaction},
{alarm,close,access}, {alarm,close,open}, {alarm,gettimeofday,alarm},
{read,uname,brk}, {read,rt_sigaction,close}, {read,fstat64,mmap},
{read,fcntl64,rt_sigaction}, {read,read,uname},
{access,setuid32,setreuid32}, {read,read,read}, {access,open,fcntl64},
{access,open,alarm}, {read,read,close}, {read,write,_llseek},
{read,write,read}, {read,write,write}, {read,write,ioctl},
```

```
{read,open,fstat64}, {read,open,fcntl64}, {read,close,alarm},
{read,close,munmap}, {write,time,rt_sigaction}, {write,_llseek,time},
{write,fstat64,ioctl}, {write,fcntl64,rt_sigaction},
{write,read,rt_sigaction}, {write,read,write}, {write,read,open},
{write,write,fstat64}, {write,write,read}, {write,brk,time},
{brk,time,rt_sigaction}, {brk,time,open}, {brk,brk,brk},
{brk,brk,setpriority}, {write,ioctl,close}, {brk,open,open},
{brk,readlink,setpgid}, {brk,setpriority,uname}, {write,close,socketcall},
{open,_llseek,read}, {open,rt_sigaction,read}, {open,fstat64,mmap},
{open,fcntl64,fcntl64}, {open,alarm,rt_sigaction}, {open,read,fstat64},
{open,read,read}, {ioctl,socketcall,socketcall},
{ioctl,rt_sigaction,vhangup}, {ioctl,fstat64,ioctl}, {ioctl,chown32,chmod},
{ioctl,brk,readlink}, {ioctl,ioctl,fstat64}, {ioctl,close,munmap},
{ioctl,mmap,read}, {ioctl,mmap,write}, {open,open,fstat64},
{open,open,read}, {open,ioctl,ioctl}, {open,open,setpriority},
{setpgid,ioctl,chown32}, {open,setpriority,open},
{close,socketcall,socketcall}, {close,stat64,rt_sigaction},
{close,alarm,getuid32}, {close,access,open}, {dup2,dup2,close},
{dup2,dup2,dup2}, {dup2,close,ioctl}, {close,open,_llseek},
{close,open,fstat64}, {close,open,fcntl64}, {close,open,read},
{close,ioctl,socketcall}, {close,ioctl,fstat64}, {close,ioctl,brk},
{close,dup2,dup2}, {setsid,open,fcntl64}, {close,close,ioctl},
{close,close,close}, {close,close,dup2}, {close,mmap,munmap},
{close,munmap,uname}, {close,munmap,rt_sigaction}, {close,munmap,fstat64},
{close,munmap,setgroups32}, {close,munmap,chown32}, {close,munmap,open},
{close,munmap,setpriority}, {gettimeofday,alarm,rt_sigaction},
{readlink,setpgid,ioctl}, {mmap,mprotect,mmap}, {mmap,_llseek,read},
{mmap,read,read}, {mmap,read,open}, {mmap,read,close},
{mmap,write,_llseek}, {mmap,write,read}, {mmap,write,write},
{mmap,write,brk}, {mmap,close,open}, {mmap,close,mmap},
{mmap,close,munmap}, {mmap,mmap,close}, {mmap,munmap,rt_sigaction},
{munmap,exit,uname}, {munmap,uname,open}, {munmap,rt_sigaction,socketcall},
{munmap,rt_sigaction,alarm}, {munmap,fstat64,ioctl},
{munmap,setgroups32,open}, {munmap,chown32,chmod}, {munmap,open,fstat64},
{munmap,open,fcntl64}, {munmap,open,setpriority},
{munmap,setpriority,rt_sigaction}, {munmap,setpriority,fstat64},
{munmap,setpriority,write}, {setpriority,uname,ugetrlimit},
{setpriority,rt_sigaction,rt_sigaction}, {setpriority,fstat64,ioctl},
{setpriority,write,time}, {setpriority,write,brk}, {setpriority,open,ioctl}
```

Tabela B.2 – Seqüências de chamadas de sistema do comando login alterado

```
{socketcall,socketcall,rt_sigaction}, {socketcall,socketcall,close},
{socketcall,rt_sigaction,write}, {socketcall,rt_sigaction,open},
{socketcall,fcntl64,socketcall}, {socketcall,close,open},
{execve,wait4,rt_sigaction}, {vhangup,rt_sigaction,setsid},
{execve,uname,brk}, {exit,uname,brk}, {wait4,rt_sigaction,rt_sigaction},
{chdir,execve,uname}, {uname,ugetrlimit,close}, {uname,brk,open},
{uname,open,fcntl64}, {mprotect,mmap,close}, {mprotect,mmap,mmap},
{time,rt_sigaction,socketcall}, {time,write,close}, {time,open,fstat64},
{_llseek,time,write}, {_llseek,alarm,rt_sigaction}, {_llseek,read,read},
{_llseek,read,write}, {_llseek,read,open}, {_llseek,read,close},
{_llseek,write,fcntl64}, {_llseek,write,close}, {chmod,chmod,setgid32},
{chmod,fstat64,chown32}, {chmod,setgid32,brk},
{chmod,ioctl,rt_sigaction}, {rt_sigaction,socketcall,rt_sigaction},
{rt_sigaction,socketcall,fcntl64}, {rt_sigaction,vhangup,rt_sigaction},
{rt_sigaction,rt_sigaction,rt_sigaction},
{rt_sigaction,rt_sigaction,rt_sigprocmask},
{rt_sigaction,rt_sigaction,setuid32}, {rt_sigaction,rt_sigaction,brk},
{rt_sigaction,rt_sigprocmask,exit}, {rt_sigaction,rt_sigprocmask,execve},
{rt_sigaction,rt_sigprocmask,fork}, {rt_sigaction,rt_sigprocmask,munmap},
{rt_sigaction,setuid32,chdir}, {rt_sigaction,alarm,rt_sigaction},
{rt_sigaction,alarm,fcntl64}, {rt_sigaction,alarm,alarm},
{rt_sigaction,alarm,close}, {rt_sigaction,alarm,gettimeofday},
{rt_sigaction,read,write}, {rt_sigaction,write,write},
{rt_sigaction,brk,brk}, {rt_sigaction,open,rt_sigaction},
{rt_sigaction,close,stat64}, {rt_sigaction,setsid,open},
{rt_sigprocmask,execve,wait4}, {rt_sigprocmask,exit,uname},
{rt_sigprocmask,fork,rt_sigaction}, {rt_sigprocmask,munmap,exit},
{ugetrlimit,close,close}, {stat64,rt_sigaction,rt_sigaction},
{fstat64,_llseek,write}, {fstat64,chown32,chown32}, {fstat64,ioctl,mmap},
{fstat64,mmap,mprotect}, {fstat64,mmap,_llseek}, {fstat64,mmap,fstat64},
```

```
{fstat64,mmap,read}, {fstat64,mmap,close},
{getuid32,getegid32,setregid32}, {getegid32,setregid32,setreuid32},
{setreuid32,setregid32,getpid}, {setreuid32,access,setuid32},
{getpid,access,open}, {setregid32,getpid,access},
{setregid32,setreuid32,access}, {setgroups32,open,open},
{chown32,chmod,chmod}, {chown32,chmod,fstat64}, {chown32,chmod,ioctl},
{chown32,chown32,chmod}, {setuid32,chdir,execve},
{setuid32,setreuid32,setregid32}, {setgid32,brk,time},
{fork,rt_sigaction,rt_sigaction}, {fcntl64,socketcall,socketcall},
{fcntl64,_llseek,alarm}, {fcntl64,_llseek,write},
{fcntl64,rt_sigaction,alarm}, {fcntl64,fstat64,mmap},
{fcntl64,fcntl64,_llseek}, {fcntl64,fcntl64,fstat64},
{fcntl64,fcntl64,close}, {fcntl64,read,fcntl64}, {fcntl64,close,close},
{alarm,rt_sigaction,rt_sigaction}, {alarm,rt_sigaction,alarm},
{alarm,getuid32,getegid32}, {alarm,fcntl64,_llseek},
{alarm,fcntl64,read}, {alarm,alarm,rt_sigaction}, {alarm,close,access},
{alarm,close,open}, {alarm,gettimeofday,alarm}, {read,uname,brk},
{read,rt_sigaction,close}, {read,fstat64,mmap},
{read,fcntl64,rt_sigaction}, {access,setuid32,setreuid32},
{read,read,read}, {access,open,fcntl64}, {access,open,alarm},
{read,read,close}, {read,write,_llseek}, {read,write,read},
{read,write,write}, {read,write,ioctl}, {read,open,fstat64},
{read,open,fcntl64}, {read,close,alarm}, {read,close,munmap},
{write,time,rt_sigaction}, {write,_llseek,time}, {write,fstat64,ioctl},
{write,fcntl64,rt_sigaction}, {write,read,rt_sigaction},
{write,read,write}, {write,read,open}, {write,write,fstat64},
{write,write,read}, {write,brk,time}, {brk,time,rt_sigaction},
{brk,time,open}, {brk,brk,brk}, {brk,brk,setpriority},
{write,ioctl,close}, {brk,open,open}, {brk,readlink,setpgid},
{brk,setpriority,uname}, {write,close,socketcall}, {write,close,munmap},
{open,_llseek,read}, {open,rt_sigaction,read}, {open,fstat64,mmap},
{open,fcntl64,fcntl64}, {open,alarm,rt_sigaction}, {open,read,fstat64},
```

```
{open,read,read}, {ioctl,socketcall,socketcall},
{ioctl,rt_sigaction,vhangup}, {ioctl,fstat64,ioctl},
{ioctl,chown32,chmod}, {ioctl,brk,readlink}, {ioctl,ioctl,fstat64},
{ioctl,close,munmap}, {ioctl,mmap,read}, {ioctl,mmap,write},
{open,open,fstat64}, {open,open,read}, {open,ioctl,ioctl},
{open,open,setpriority}, {setpgid,ioctl,chown32},
{open,setpriority,open}, {close,socketcall,socketcall},
{close,stat64,rt_sigaction}, {close,alarm,getuid32}, {close,access,open},
{dup2,dup2,close}, {dup2,dup2,dup2}, {dup2,close,ioctl},
{close,open,_llseek}, {close,open,fstat64}, {close,open,fcntl64},
{close,open,read}, {close,ioctl,socketcall}, {close,ioctl,fstat64},
{close,ioctl,brk}, {close,dup2,dup2}, {setsid,open,fcntl64},
{close,close,ioctl}, {close,close,close}, {close,close,dup2},
{close,mmap,munmap}, {close,munmap,uname}, {close,munmap,rt_sigaction},
{close,munmap,fstat64}, {close,munmap,setgroups32},
{close,munmap,chown32}, {close,munmap,open}, {close,munmap,setpriority},
{gettimeofday,alarm,rt_sigaction}, {readlink,setpgid,ioctl},
{mmap,mprotect,mmap}, {mmap,_llseek,read}, {mmap,fstat64,_llseek},
{mmap,read,uname}, {mmap,read,read}, {mmap,read,write},
{mmap,read,open}, {mmap,read,close}, {mmap,write,_llseek},
{mmap,write,read}, {mmap,write,write}, {mmap,write,brk},
{mmap,close,open}, {mmap,close,mmap}, {mmap,close,munmap},
{mmap,mmap,close}, {mmap,munmap,rt_sigaction}, {munmap,exit,uname},
{munmap,uname,open}, {munmap,rt_sigaction,socketcall},
{munmap,rt_sigaction,rt_sigaction}, {munmap,rt_sigaction,alarm},
{munmap,fstat64,ioctl}, {munmap,setgroups32,open},
{munmap,chown32,chmod}, {munmap,open,fstat64}, {munmap,open,fcntl64},
{munmap,open,setpriority}, {munmap,setpriority,fstat64},
{munmap,setpriority,write}, {munmap,setpriority,open},
{setpriority,uname,ugetrlimit}, {setpriority,fstat64,ioctl},
{setpriority,write,time}, {setpriority,write,brk},
{setpriority,open,fstat64}, {setpriority,open,ioctl}
```