FÁBIO ANDRÉ SEIDEL

PROVENDO COMPORTAMENTO DISCRICIONÁRIO AO MODELO BIBA DE INTEGRIDADE MULTINÍVEL (LOW-WATERMARK)

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

2005

FÁBIO ANDRÉ SEIDEL

PROVENDO COMPORTAMENTO DISCRICIONÁRIO AO MODELO BIBA DE INTEGRIDADE MULTINÍVEL (LOW-WATERMARK)

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: *Metodologias e Técnicas de Computação*

Orientador: Prof. Dr. Altair Olivo Santin

CURITIBA

2005

	Seidel, Fábio André
S458p	Provendo Comportamento Discricionário ao Modelo BIBA de
2005	Integridade Multinível (low-watermark) / Fábio André Seidel :
	orientador, Altair Olivo Santin. – 2005.
	xv, 104 f. : il. ; 30 cm
	Dissertação (Mestrado) – Pontifícia Universidade Católica do Paraná, Curitiba, 2005. Inclui bibliografia
	1. Sistemas operacionais (Computadores). 2. Redes de computação - Medidas de segurança. 3. Computadores – Controle de acesso. I. Santin, Altair Olivo. II. Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática Aplicada. III. Título.
	CDD 20. ed. 005.42
	004. 60289

Dedico este trabalho a minha esposa.

Agradecimentos

Primeiramente quero agradecer ao meu amigo José Maciel Pereira pela oportunidade proporcionada para realização deste trabalho, e dizer que seu apoio foi fundamental para a conclusão do mesmo.

Agradeço a minha esposa Elaine, que sempre teve compreensão nos momentos mais difíceis. Aos meus amigos Uriel e Leonardo que me auxiliaram no decorrer deste trabalho.

Agradeço ao Altair Olivo Santin pelo apoio, compreensão e orientação no decorrer do desenvolvimento deste trabalho.

Sumário

Agradecimentos .... **v**
Sumário .... **vi**
Lista de Figuras .... **x**
Lista de Tabelas .... **xi**
Lista de Abreviaturas .... **xii**
Resumo .... **xiii**
Abstract .... **xv**

CAPÍTULO 1 **1**
Introdução 1
1.1. Motivação 1
1.2. Proposta 2
1.3. Organização do trabalho 4

CAPÍTULO 2 **5**
Fundamentos de Segurança em Sistemas Computacionais 5
2.1. Conceito de segurança 5
2.2. Violação de segurança 7
2.3. Políticas 8
2.4. Modelos de segurança 9
2.4.1. Matriz de acesso 10
2.4.2. Modelo discricionário 11
2.4.3. Modelo obrigatório 12
2.4.4. Modelo Bell e Lapadulla 12
2.4.5. Modelo de integridade BIBA 13
2.4.6. Modelo baseado em papéis 15
2.5. Mecanismos de segurança 16
2.5.1. Controle de acesso discricionário 17
2.5.2. Controle de acesso obrigatório 17
2.5.3. Controle de acesso baseado em papéis 18

2.6. Conclusão.. 18

CAPÍTULO 3 **19**

Proteção de Integridade do Sistema Operacional.. 19

3.1. Proteção de integridade.. 19

3.1.1. Funções *Hash*.. 19

3.1.1.1. Funções de *hash* customizadas... 21

3.1.1.2. Message Digest 4 (MD4).. 21

3.1.1.3. Secure Hash Algorithm (SHA1)... 22

3.2. Sistemas Clássicos.. 23

3.2.1. Lista de controle de acesso.. 24

3.2.2. Capabilities.. 25

3.3. Proteção de integridade *batch* (*offline*)... 26

3.3.1. Tripwire.. 26

3.3.2. Advanced Intrusion Detection Environment (AIDE)....................... 28

3.4. Sistemas adicionais de proteção... 28

3.4.1. Linux Intrusion Detection System (LIDS)...................................... 28

3.4.2. Linux Security Modules (LSM).. 29

3.4.3. Verificação de integridade de arquivos pelo kernel do sistema operacional.. 31

3.5. Conclusão... 34

CAPÍTULO 4 **35**

Mecanismos de Controle de Acesso Obrigatório.. 35

4.1. Modelo do cofre... 35

4.1.1. Cofre de usuário.. 36

4.1.2. Cofre global.. 36

4.1.3. Cofre escrow (Garantia).. 37

4.1.4. Cofre fundamental.. 37

4.1.5. Proteção de arquivos... 37

4.1.6. Vantagens dos cofres... 38

4.2. Modelo de autorização multi-nível.. 38

4.2.1. Servidor de autorização.. 38
4.2.2. Segurança do kernel.. 39
4.2.3. Controle dos direitos de acesso.. 40
4.2.4. Aplicação do modelo multi-nível.. 40
4.3. Máquinas virtuais.. 42
4.4. SELinux... 43
4.4.1. Servidor de segurança.. 45
4.5. LOMAC (*Low water mark*)... 45
4.5.1. Proteção de integridade.. 47
4.6. Separação de privilégios de execução.. 48
4.6.1. Menor privilégio.. 48
4.6.2. Separação de privilégio.. 49
4.6.3. Separação de privilégios SSH... 50
4.7. Outros mecanismos que podem ser utilizados para prover segurança.......... 51
4.7.1. Janus.. 52
4.7.2. Kernel Hypervisors.. 52
4.7.3. Generic Software Wrappers Toolkit... 52
4.7.4. Medusa DS9.. 52
4.7.5. Remus... 53
4.7.6. VXE... 53
4.8. Conclusão.. 53

CAPÍTULO 5 **54**

Provendo Comportamento Discricionário ao Modelo BIBA de Integridade.......... 54
5.1. Problema... 54
5.2. Proposta.. 57
5.3. Recursos utilizados na proposta.. 58
5.3.1. Repositório de registros... 58
5.3.2. Dispositivo de transição.. 59
5.3.3. Serviço de Log... 59
5.4. Dinâmica do mecanismo proposto.. 60
5.4.1. Dispositivo baseado em OSE... 61

5.4.2. Serviço KTlog 62
5.5. Detalhamento estruturado da proposta 62
5.6. Problema da troca de senhas 64
5.6.1. Proposta para troca de senhas 64
5.7. Considerações sobre a proposta 65
5.7.1 Benefícios da proposta 66
5.7.2 Limitações da proposta 67
5.8. Conclusão 67

CAPÍTULO 6 **69**
Implementação e Resultados 69
6.1. Alteração do código do LOMAC 69
6.2. Alteração do arquivo de configuração do *syslog* 70
6.3. Implementação do dispositivo baseado em OSE 71
6.4. Implementação do KTlog 71
6.5. Avaliação do modelo proposto 75
6.5.1. Desempenho do KTlog *read/write* 75
6.5.2. Desempenho do KTlog *sendfile* 76
6.5.3. Avaliação de desempenho do protótipo 77
6.5.4. Testes com a utilização de *exploit* 77
6.5.5. Testes de parada do serviço de registro 77
6.6. Análise dos resultados obtidos 78
6.7. Conclusão 78

CAPÍTULO 7 **80**
Conclusão 80

Referências Bibliográficas 84

Lista de Figuras

Figura 2.1.	Matriz de Acesso..	10
Figura 2.2.	Regras do Modelo Obrigatório de Integridade BIBA......................	14
Figura 2.3.	Modelo BIBA da Marca D`Água Baixa do Sujeito.........................	15
Figura 2.4.	Modelo RBAC Básico...	15
Figura 3.1.	Modelo da Lista de Controle de Acesso..	24
Figura 3.2.	Modelo de Capabilities..	25
Figura 3.3.	Arquitetura LSM Hooks...	30
Figura 3.4.	LSM hook de permissão..	31
Figura 3.5.	Representação do Monitor de Referência.......................................	32
Figura 4.1.	Modelo LOMAC com 2 níveis de integridade................................	46
Figura 5.1.	Funcionamento do LOMAC para o *syslogd*..................................	57
Figura 5.2.	Repositório de nível de integridade 3 armazenando dados de nível de integridade 1 e 2..	58
Figura 5.3.	Modelo de utilização do dispositivo baseado em OSE....................	61
Figura 5.4.	Modelo de utilização do serviço KTlog...	62
Figura 5.5.	Modelo Geral da Proposta...	63
Figura 5.6.	Modelo Proposto para troca de senhas..	65
Figura 6.1.	Chamadas de sistema utilizadas pelo KTlog...................................	72
Figura 6.2.	Função de utilização da chamada de sistema *sendfile*.....................	73
Figura 6.3.	Função de utilização da Kernel *Thread*..	74

Lista de Tabelas

Tabela 6.1. Tempo e Consumo do KTlog Read/Write.. 75

Tabela 6.2. Tempo e Consumo do KTlog Sendfile... 76

Tabela 6.3. Desempenho do Protótipo.. 76

Lista de Abreviaturas

ACL	Access Control List
AIDE	Advanced Intrusion Detection Environment
API	Application Program Interface
BLP	Bell Lapadulla
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DAC	Discricionary Access Control
FLASK	Flux Advanced Security Kernel
FTP	File Transfer Protocol
GID	Group Identity
GPL	General Public License
IDS	Intrusion Detection Service
LIDS	Linux Intrusion Detection System
LKM	Loadable Kernel Module
LOMAC	Low Water Mark Mandatory Access Control
LSM	Linux Security Module
MAC	Mandatory Access Control
MD	Message Digest
NFS	Network File System
NSA	National Security Agency
PGP	Pretty Good Privacy
RBAC	Role Based Access Control
SHA	Security Hash
SSH	Secure Shell
SSL	Secure Sockets Layer
TE	Type Enforcement
UID	User Identity
VM	Virtual Machine

Resumo

A complexidade dos sistemas operacionais comerciais atuais, faz com que erros em sua programação sejam cometidos. A exposição destes sistemas operacionais à Internet os torna vulneráveis e suscetíveis a ataques. Os ataques por intrusos, cavalos de tróia ou vírus demandam um crescente desenvolvimento de mecanismos de proteção aos pontos fundamentais de segurança (integridade, confidencialidade e disponibilidade). Estes mecanismos podem comprometer o desempenho ou acrescentar mais pontos de vulnerabilidade. A utilização do modelo obrigatório baseado em políticas mais rígidas e inflexíveis procura dificultar a violação dos controles visando uma proteção mais confiável dos sistemas operacionais. A implementação de políticas multinível (baseadas no modelo obrigatório) nos sistemas operacionais comerciais, geralmente é efetivada com algum tipo de relaxamento das regras da política. Este tipo de relaxamento caracteriza com algum nível de intensidade a compatibilidade da implementação com as regras do modelo. Tal relaxamento é necessário para viabilizar a utilização de políticas multinível em sistemas operacionais de propósito geral, baseados em políticas discricionárias. Este trabalho visa apresentar extensões ao mecanismo de controle de integridade multinível, LOMAC (implementação para Linux), para torná-lo mais compatível com o modelo de integridade multinível do BIBA. A proposta oferece uma alternativa aos relaxamentos mais intensos feitos nas regras de política multinível na implementação do LOMAC no Linux. As extensões propostas permitem que um sujeito de baixo nível de integridade faça a escrita seqüencial de registros (string de caracteres) - com o mesmo nível de integridade - em um container de alto nível de integridade. Além disto, é proposta a uma outra extensão que permite a um sujeito de baixo nível de integridade a leitura e a atualização (escrita) de registros – de mesmo nível de integridade – em um container de alto nível de integridade. As extensões propostas estão em acordo com as regras do modelo multinível de integridade do BIBA. Na prática até os autores do modelo obrigatório admitiam a existência de sujeitos de confiança para manter o sistema operável, nesta proposta tais sujeitos não são necessários para que o sistema continue seguro e operável; A proposta mantém o comportamento do sistema inalterado para programas executáveis.

Palavras-chave: Políticas de segurança multi-nível, mecanismo de controle de integridade obrigatório, Integridade em nível de sistema operacional, *Low-Watermark Mandatory Access Control.*

Abstract

The complexity of the current commercial operating systems makes the programming errors more susceptible. The exposition of these operational systems makes it vulnerable and susceptible to attacks. The attacks made by intruders, such as Trojan or virus, demands an constant development of mechanisms for protection, to the main security issues, such as integrity, confidentiality and availability. These mechanisms may compromise the performance, or adds more vulnerability to the system. The use of mandatory model, based on rigid and inflexible policies, try to make difficult the violation of the system control, looking for a protection level more reliable to the operating system. The implementation of multi-level policies on current operational systems, based on the mandatory model, usually is done by relaxing the policies rules. This relaxing is required to make possible the using of multi-level policies, on operational systems for general purpose, based on discretionary policies. This work introduces the extensions of multi-level integrity control mechanism, LOMAC (for Linux implementation), to make it more compatible to the BIBA multi-level integrity model. This purpose offers an alternative to the more intense relaxation, on the multi-level policies rules, to the LOMAC implementation for Linux. The purposed extensions allows to a subject, on the low level integrity, to write sequentially on records – with the same level of integrity – in a high level integrity container. Another extension is introduced, that allows a low level integrity process to read and write on records – in the same level of integrity– in a high level integrity container.The purposed extensions are in agreement with the rules to the BIBA multi-level integrity model. Practically, even the authors of the mandatory model admit the existence of the trusted subject, to keep the system liable. In this purpose, these subjects are not required to keep the system safe and operational; the purpose maintains the behavior of the system unchangeable to the executable programs.

Keywords: Multi-level Security Policies; Mandatory Integrity Control Mechanism; Operational System Integrity; Low-Watermark Mandatory Access Control.

Capítulo 1

Introdução

1.1. Motivação

A utilização da Internet expõe os sistemas operacionais e aplicações ao risco, exigindo uma postura diferente dos especialistas na concepção de segurança para este ambiente distribuído. Os riscos de ataques[1] a pontos falhos/frágeis (vulneráveis) dos sistemas e/ou a aplicações comerciais podem afetar as propriedades fundamentais de um sistema de segurança (integridade, confidencialidade e disponibilidade). Constantemente novas soluções são oferecidas para prevenção contra ataques que podem comprometer a segurança do sistema ou das aplicações, porém a inserção de novos componentes de software pode criar outros pontos vulneráveis no ambiente.

Os sistemas operacionais comerciais expõem suas possíveis vulnerabilidades a partir do momento que estão em funcionamento, seja em computadores conectados em redes locais ou mesmo em computadores sem qualquer acesso a um ambiente distribuído. Mas, quando os mesmos são conectados a um ambiente distribuído é criada uma porta de comunicação com possíveis intrusos virtuais. Desta forma, faz-se necessária a utilização de políticas de controle de acesso, que na maioria das vezes são baseadas no modelo discricionários. Onde o usuário é identificado por um nome de usuário e uma senha. O modelo obrigatório utiliza políticas multinível [AMO94] – separando o sistema em diferentes níveis de acesso – para restringir o

[1] Ataques são ações de malfeitores lançadas contra um sistema alvo visando causar-lhe algum tipo de dano.

acesso não autorizado as informações, mas devido ao fato de ter regras inflexíveis, necessita do relaxamento dessas regras para ser viabilizado nos sistemas operacionais comerciais.

A utilização de políticas multinível no controle de acesso em sistemas operacionais comercias necessita do relaxamento de suas regras, devido a certa incompatibilidade operacional entre tal sistema e o comportamento dos usuários dos sistemas comerciais. Este relaxamento se faz necessário para tornar operável o sistema obrigatório (multinível) criando diferenças significativas entre modelo e tal sistema.

O nível de relaxamento das regras da política multinível determina sua compatibilidade com o modelo obrigatório. As implementações do modelo obrigatório de integridade do BIBA [AMO94, BIB77] sofrem problemas de compatibilidade principalmente na escrita seqüencial de registros de baixo nível de integridade em arquivos de alto nível de integridade. A mesma incompatibilidade é notada quando se faz necessária à leitura e a alteração de registros de baixa integridade a partir de arquivos de alta integridade. É importante notar que em ambos os casos os registro de baixa integridade são gerados ou alterados pelos proprietários dos mesmos. Logo, por mais que as regras da política sejam coerentes, não é aceitável que o proprietário de registros com menor nível de integridade não consiga armazená-los em um container com maior nível de integridade, objetivando a preservação de sua integridade. Este trabalho é instigado pelo desejo de propor extensões ao modelo de integridade obrigatório com o intuito de manter a proposta mais compatível que as implementações atuais.

1.2. Proposta

A proposta deste trabalho é conceber uma alternativa para que implementações do modelo de controle de acesso obrigatório visando a integridade deixem de ter limitações na sua efetivação em sistemas operacionais comerciais. As implementações mais recentes do modelo de integridade obrigatório possuem relaxamentos das regras impostas pelo modelo de BIBA [AMO94, BIB77]. A realização destes relaxamentos em aplicações tornará tais aplicações restritivas, já que a sua implementação não obedece integralmente às regras impostas pelo modelo conceitual.

A alternativa criada para se tratar a implementação do modelo de integridade obrigatório, visa a comunicação entre sujeitos e objetos de diferentes níveis de integridade, sem que sejam violadas as regras impostas pelo modelo de BIBA. Para elaborar o modelo proposto (onde um sujeito de alto nível de integridade possa realizar a leitura de um objeto com baixo nível de integridade), necessitou-se implementar dois novos mecanismos que são utilizados na concepção do modelo. Deste modo, o primeiro mecanismo utilizado terá a funcionalidade de um objeto de transição (memória temporária), entre um nível inferior de integridade e um nível superior de integridade. O segundo mecanismo implementado terá a funcionalidade de realizar a leitura de um objeto de transição – com nível de integridade irrelevante – e escrever as informações lidas em um local de armazenamento de informações com nível de integridade mais elevado.

Propõe-se também, fazer com que um sujeito com um menor nível de integridade possa alterar informações que sejam pertinentes a si e que estejam armazenadas em um local que possua um nível de integridade mais elevado. Para isto, pretende-se fazer a utilização de uma técnica de programação que visa separar o código executando operações privilegiadas de uma aplicação do código executando operações não privilegiadas [PRO03]. O objetivo é viabilizar que um sujeito com um menor nível de integridade realize alteração de um registro (com informações que lhe pertencem) contido em um local de armazenamento com um nível de integridade mais elevado. Neste caso, o código privilegiado efetiva a operação de modificação (escrita) de parte do conteúdo textual de um registro pertencente ao sujeito a que o registro se refere. Não se pretende alterar o comportamento do modelo de integridade obrigatório com relação a operações de leitura e escrita de conteúdos binários (como programas executáveis).

As extensões propostas dão um certo "nível de liberdade" ao dono dos registros de informação textual para alterá-las ou mesmos para gerá-los em diferentes níveis de integridade. Visa também estabelecer um maior nível de proteção de integridade, similar ao que acontece no sistema discricionário. Porém, as extensões propostas, mantém a política de integridade multinível, ou seja, não violando as regras do modelo de integridade do BIBA.

1.3. Organização do Trabalho

Este trabalho é composto de 7 capítulos, incluindo este. O capítulo 2 apresenta os fundamentos de segurança em sistemas computacionais; o capítulo 3 conceitua e detalha os mecanismos de proteção de integridade em nível de sistema operacional; o capítulo 4 conceitua e detalha os mecanismos de controle de acesso obrigatório; o capítulo 5 contém a proposta da dissertação e os conceitos relacionados à mesma; o capítulo 6 apresenta o protótipo implementado e os testes realizados com a análise crítica dos resultados obtidos; e por último o capítulo 7 conclui a dissertação, apresentando os trabalhos futuros relacionados à proposta e a sua implementação.

Capítulo 2

Fundamentos de segurança em sistemas computacionais

Com a interligação dos computadores, através de redes físicas ou *wireless* (conexões de equipamentos sem fio), estes computadores se tornam mais suscetíveis a acessos não autorizados e o conceito de segurança, importante em ambientes físicos se torna relevante para ambientes distribuídos. Desta forma é necessário que os usuários de um sistema computacional tenham acesso somente ao que lhe seja pertinente. Pois caso isso não ocorra, o sistema se torna mais suscetível a ataques podendo causar sérios danos ao sistema.

2.1. Conceito de segurança

A segurança de sistemas computacionais visa à proteção contra a indisponibilidade, vazamento da informação e a leitura ou modificação não-autorizada das informações. Além de garantir dados e informações, a segurança visa prevenir, detectar, conter e documentar eventuais ameaças aos sistemas computacionais [BIS03].

Um sistema computacional pode ser considerado seguro se este atender a três requisitos básicos: confidencialidade, integridade e disponibilidade [DEN82].

- **Confidencialidade:** A informação poderá ser acessada somente por usuários autorizados. Este acesso é restrito a usuários devidamente cadastrados para que impeça usuários não autorizados de terem acesso à informação.

- **Integridade:** Prover a garantia que a informação deve ser retornada em sua forma original no momento em que foi armazenada, protegendo contra modificações não autorizadas;
- **Disponibilidade:** A informação ou sistema de computador deve estar disponível no momento em que a mesma seja requisitada.

Alguns autores tratam que um sistema será seguro se abordar outros dois itens: Autenticidade e Não repúdio [BIS03, LAN01].

- **Autenticidade** – Garante que a informação ou o usuário da mesma é autêntico; Atesta a origem do dado ou informação;
- **Não repúdio** – Não é possível negar (no sentido de dizer que não foi feito) uma operação ou serviço que modificou ou criou uma informação;

Outros três itens podem ser desejáveis para se ter um ambiente seguro: Legalidade, Privacidade e Auditoria [BIS03].

- **Legalidade** – Garante a legalidade (jurídica) da informação; Aderência de um sistema à legislação; Característica das informações que possuem valor legal dentro de um processo de comunicação, onde todos os ativos estão de acordo com as cláusulas contratuais pactuadas ou a legislação política institucional, nacional ou internacional vigentes.
- **Privacidade** – Uma informação pode ser considerada confidencial, mas não privada. Uma informação privada deve ser vista/lida/alterada somente pelo seu proprietário. Consiste em garantir que a informação não será disponibilizada para outras pessoas (neste caso é atribuído o caráter de confidencialidade a informação);
- **Auditoria** – Verificar e mapear os passos que um determinado processo realizou ou que uma informação foi submetida, identificando os participantes, os locais e horários de cada etapa. Auditoria em software significa uma parte da aplicação, ou conjunto de funções do sistema, que viabiliza uma auditoria, consistindo no exame do histórico dos eventos dentro de um sistema para determinar quando e onde ocorreu uma violação de segurança;

2.2. Violação de segurança

Uma ameaça consiste em uma possível violação da segurança de um sistema. O termo ameaça é utilizado para identificar circunstâncias, condições ou eventos que forneçam algum potencial de violação de segurança. Quando se faz referência ao termo vulnerabilidade, isto quer dizer que este termo está se referindo a falhas ou características que podem ser exploradas em determinados sistemas computacionais. Através destas vulnerabilidades, sistemas podem sofrer violações e estas podem ser divididas em três categorias:

- Modificações ou deturpação da informação
    - Violações que atentam contra a integridade das informações
- Revelação não autorizada de informação
    - Verificação da propriedade de confidencialidade
- Negação ou interrupção de serviços
    - Impedimento do acesso legítimo aos recursos para um usuário autorizado

A concretização das violações varia desde a observação de dados com ferramentas simples de monitoramento de redes, a ataques sofisticados baseados no conhecimento do funcionamento do sistema. Um ataque é identificado como um conjunto de ações conduzidas por uma entidade não autorizada, tendo como objetivo a violação de segurança [BIS03].

Ataques passivos são os que, quando realizados, não resultam em qualquer modificação nas informações contidas em um sistema, em sua operação ou em seu estado. Uma realização de um ataque ativo a um sistema envolve a alteração da informação contida no mesmo, ou modificações em seu estado ou operação. Alguns dos principais ataques que podem ocorrer em um ambiente de processamento e comunicação de dados são os seguintes:

- **Personificação (*masquerade*)**: uma entidade faz-se passar por outra. Uma entidade que possui poucos privilégios pode fingir ser outra, para obter privilégios extras;

- ***Replay (ataque de mensagem antiga)**:* Uma mensagem, ou parte dela é interceptada, e posteriormente transmitida para produzir um efeito não autorizado. Por exemplo, uma mensagem válida, carregando informações que autenticam uma entidade A, pode ser capturada e posteriormente transmitida por uma entidade X tentando autenticar-se no sistema (possivelmente personificando a entidade A);
- **Modificação:** o conteúdo de uma mensagem é alterado implicando em efeitos não autorizados sem que o sistema consiga detectar a alteração;
- **Recusa ou Impedimento de Serviço:** ocorre quando uma entidade não executa sua função apropriadamente ou atua de forma a impedir que outras entidades executem suas funções. Uma entidade pode utilizar essa forma de ataque para suprimir as mensagens, por exemplo, direcionadas à entidade encarregada da execução do serviço de auditoria de segurança;
- **Ataques Internos:** ocorrem quando usuários legítimos comportam-se de modo não autorizado ou não esperado;
- **Armadilhas (*trapdoor*):** ocorre quando uma entidade do sistema é modificada para produzir efeitos não autorizados em resposta a um comando (emitido pela entidade que está atacando o sistema) ou a um evento, ou seqüência de eventos, predeterminado;
- **Cavalos de Tróia:** nesse ataque, uma entidade executa funções não autorizadas, em adição às que está autorizada a executar. Um procedimento de login modificado, que, além de sua função normal de iniciar a sessão de trabalho dos usuários, grava suas senhas em um arquivo desprotegido, é um exemplo de Cavalo de Tróia.
- **Buffer Overflow:** Estouro de Buffer. Quando a memória secundária (Buffer) recebe mais dados do que pode suportar, desta maneira o sistema se torna suscetível.

2.3. Políticas

Quando um administrador de um ambiente computacional necessita tomar decisões para realizar a proteção do ambiente computacional, o mesmo necessita estabelecer regras; definindo as funcionalidades que irá oferecer, e qual será a facilidade de utilizá-la. Às vezes se torna complicado realizar decisões sobre segurança, sem que tenham determinado quais são as suas metas de segurança.

O estabelecimento de um conjunto de regras irá definir as políticas de segurança, o principal objetivo de uma política de segurança é informar aos usuários, equipe e gerentes, as suas obrigações para a proteção da tecnologia e do acesso à informação. Os objetivos determinados devem ser comunicados a todos os usuários, pessoal operacional, e gerentes através da política de segurança adotada.

Em um ambiente computacional as políticas de segurança são classificadas em duas categorias: discricionárias e obrigatórias [SHI00].

- Políticas discricionárias: Os acessos a cada recurso ou informação são manipulados livremente pelo proprietário ou responsável pelo mesmo, segundo a sua vontade (à sua discrição).
- Políticas obrigatórias: As autorizações de acesso são definidas através de um conjunto incontornável de regras, envolvendo a segurança das informações no sistema como um todo.

O objetivo da utilização de políticas de segurança no ambiente computacional é tornar o sistema mais seguro contra intrusos. Com a adoção de políticas de segurança tanto discricionárias quanto obrigatórias tornam o ambiente mais seguro, mas não livre de ações que tentam contornar as políticas de segurança adotadas.

2.4. Modelos de segurança

Os modelos de segurança são formas de descrever as políticas de segurança, determinando o comportamento de entidades administradas pela política e também as regras que definem a evolução desta política.

Quando alguns modelos matemáticos formais de segurança são utilizados para descrever políticas de autorização, os mesmos permitem de alguma maneira, verificações de que a política é coerente, e servem como guia para implementações de esquemas de autorização, correspondentes às especificações contidas no modelo [LAN81].

O modelo de segurança Bell e Lapadula [BEL76, AMO94] tem como foco principal a confidencialidade para descrever suas políticas; já outros modelos, como o Clark-Wilson

[CLA87] ou BIBA [BIB7] tem seu foco principal voltado à integridade. A formalização do modelo de integridade de BIBA tem destaque em avaliações de segurança de sistemas que o adotam como base para políticas obrigatórias, tendo seu exemplo no LOMAC (Low Water Mark Model Access Control) [FRA00].

2.4.1. Matriz de acesso

O modelo de proteção de matriz de acesso é representado como uma matriz padrão, onde as linhas representam domínios e as colunas representam objetos. Cada entrada na matriz consiste em um conjunto de direitos de acesso [LAM71, SIL02].

O esquema de matriz de acesso fornece um mecanismo para especificar uma série de políticas. O mecanismo consiste em implementar a matriz de acesso e garantir que as propriedades semânticas delineadas sejam mantidas.

Na Figura 2.1 é ilustrada uma matriz de acesso. Nesta matriz existem quatro domínios e quatro objetos: três arquivos (F1, F2 e F3) e uma impressora (F4). Quando um processo executa no domínio D1, o mesmo pode ler os arquivos F1 e F3. Um processo que executa no domínio D4 tem os mesmos privilégios que no domínio D1, mas, além disso, pode escrever nos arquivos F1 e F3. Observe que a impressora somente será acessada por processos que sejam executados no domínio D2.

Objeto **Domínio**	**F1**	**F2**	**F3**	**F4**
D1	Leitura		leitura	
D2				impressão
D3		Leitura	execução	
D4	leitura escrita		leitura escrita	

Figura 2.1. – Matriz de Acesso

A matriz de acesso fornece um mecanismo adequado para definir e implementar controle estrito para a associação dinâmica e estática entre processos e domínios. Quando passamos um processo de um domínio para outro, estamos executando uma operação

(mudança) em um objeto (o domínio). É possível controlar a mudança de domínio incluindo domínios entre objetos da matriz de acesso.

2.4.2. Modelo discricionário

No modelo discricionário, os direitos de acesso aos recursos ou informações são especificados para cada sujeito, pelo proprietário da informação ou recurso. As requisições para utilização de recursos são analisadas por um mecanismo de segurança discricionário e o acesso é concedido de acordo com as regras de autorização [LAN81, AMO94].

O modelo discricionário se baseia em conceder e retirar privilégios. O controle poderá ser centralizado, isto é, a autorização é administrada por um controle central, normalmente o administrador do sistema. Podem também ter um controle descentralizado de maneira que o proprietário da informação possui o direito de conceder ou retirar os privilégios.

A adoção deste modelo para aplicação em políticas de segurança tem a vantagem de torná-las flexíveis, fazendo com que tais políticas possam ser utilizadas por vários tipos de sistemas e aplicações comerciais e industriais e, conseqüentemente, atualmente serem as mais utilizadas. Vários modelos foram propostos e implementados em vários sistemas, porém certos requisitos de segurança podem não ser totalmente satisfeito, como no caso em que um usuário com permissão de leitura a determinada informação poderá transferi-la para um objeto de outro usuário que não possui esta permissão. As políticas de controle de acesso discricionárias não permitem classificar os objetos e sujeitos no que diz respeito a confidencialidade. Com isto, não é possível criar uma política que possa restringir os privilégios de acesso a informações confidenciais com base na classificação dos sujeitos, independentemente de especificações determinadas pelo administrador. Este tipo de política é importante quando se deseja limitar a autoridade do administrador, protegendo informações confidenciais, muitas vezes vitais, como no caso dos sistemas militares. Mas sua aplicação pode ser bem mais abrangente, sendo fundamental em qualquer sistema onde a confidencialidade das informações for importante.

2.4.3. Modelo obrigatório

O Modelo obrigatório tem em seu esquema de autorização um conjunto de regras incontornáveis que expressam um tipo de organização envolvendo a segurança das informações no sistema como um todo. O modelo obrigatório prevê que os usuários, objetos e recursos do sistema sejam rotulados; os rótulos dos objetos seguem uma classificação específica enquanto os usuários de acesso possuem níveis de habilitação. Os controles que determinam as autorizações de acesso são baseados numa comparação da habilitação do usuário com a classificação do objeto [AMO94].

2.4.4. Modelo Bell e Lapadulla

O modelo Bell e Lapadula (BLP) tem este nome devido aos cientistas David Bell e Leonard Lapadulla que desenvolveram este modelo no inicio da década de 70. O referido modelo é baseado nos procedimentos de manipulação de informação em áreas ligadas à segurança nacional americana [AMO94, BEL76].

A idéia principal deste modelo é acrescentar controles de acesso obrigatório aos controles de acesso discricionário, desta forma, aplicando políticas de segurança que impeçam o fluxo de informações de níveis mais altos aos níveis mais baixos de segurança. As permissões de acesso são definidas através de uma matriz de acesso e de rótulos de segurança. A matriz de acesso armazena os direitos de cada sujeito sobre os objetos do sistema e pode ser modificada pelos sujeitos através de regras específicas.

O modelo Bell-LaPadula priva a confidencialidade e está baseado na classificação dos elementos de segurança, que definem a política de acesso ao sistema. Esta classificação é expressa por níveis de segurança, onde cada nível é definido por dois componentes: uma classificação e um conjunto de categorias. O modelo BLP classifica as informações em quatro níveis hierárquicos de sensitividade (não-classificada, confidencial, secreta e ultra-secreta) obedecendo a seguinte ordem:

- US > S > C > NC.

O conjunto de categorias não possui nenhuma hierarquia e os seus elementos são definidos de acordo com o ambiente ou área de aplicação do modelo.

O modelo BLP possui duas propriedades que devem ser satisfeitas para que a segurança do sistema seja garantida.

- **Propriedade de Segurança Simples *ou No Read Up (NRU):*** Um sujeito pode ter acesso de leitura ou escrita sobre um objeto se e somente se o *clearance* do sujeito domina o nível de segurança do objeto. A propriedade de segurança simples garante que sujeitos não acessem informação acima do seu nível máximo de segurança (*clearence);*
- **Propriedade Estrela ou No Write Down (NWD):** um sujeito só pode escrever em objetos cujos níveis de segurança dominam o nível de segurança do sujeito.

Estas propriedades devem ser satisfeitas para que se possa evitar que a informação de um nível mais alto acabe fluindo para níveis baixos de segurança, caracterizando a revelação não autorizada de informação. Além das duas propriedades descritas o modelo também mantém um controle discricionário por nível de segurança (*discretionary security propriety*) que reflete os princípios de autorização expressos no modelo matriz de acesso.

2.4.5. Modelo de integridade BIBA

O modelo obrigatório de integridade BIBA [AMO94, BIB77] é descrito como o inverso do modelo BLP [AMO94]. Esta é uma descrição razoável, pois as regras básicas do modelo BIBA são bem próximas das regras do modelo BLP. No modelo de BIBA são definidas regras onde um sujeito estando em um nível de integridade mais elevado não pode ler um objeto que esteja em um nível de integridade inferior ao seu (*no read down* NRD). Também estabelece que um sujeito estando em um baixo nível de integridade não poderá escrever em um objeto em um nível de integridade superior ao seu (*no write up* NWU) [AMO94, BIB77].

O modelo obrigatório de BIBA está baseado na integridade e tanto uma regra quanto outra deste modelo, são opostas ao modelo BLP que em sua proposta visa a confidencialidade. Deste modo, os níveis mais elevados de integridade devem ser vistos como uma associação daqueles sujeitos e objetos que devem ter um nível de integridade mais

elevado, e os níveis mais baixos devem ser vistos como uma associação daqueles sujeitos e objetos que podem tolerar um menor nível de integridade.

O modelo BIBA pode ser representado através de um diagrama de níveis de maneira mais objetiva. As linhas horizontais do diagrama representam os níveis de integridade e as ações que são negadas pelo contexto geral do modelo. O diagrama do modelo BIBA pode ser melhor visualizado na Figura 2.2

.

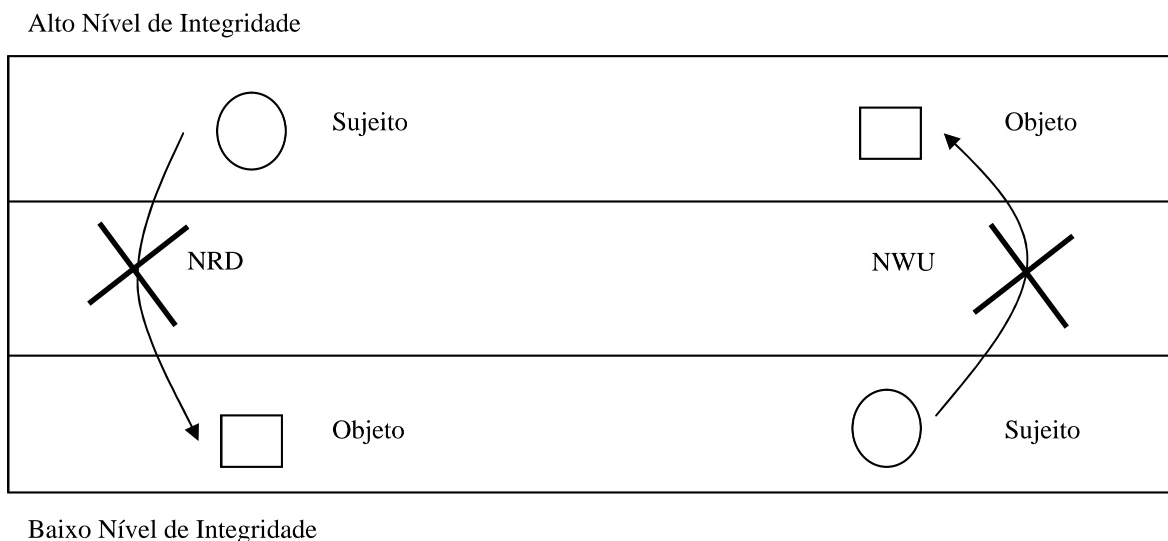

Figura 2.2. – Regras do Modelo Obrigatório de Integridade BIBA

Uma das vantagens do modelo BIBA, é a incorporação de algumas características das regras do modelo BLP incluindo a simplicidade e atributos intuitivos. Isto é, os desenvolvedores de sistemas podem facilmente entender as regras de NWD e NRU, podendo incorporá-las em projetos de decisões de sistemas.

O Modelo da marca d´água baixa do sujeito [AMO94] introduz um leve relaxamento nas regras de leitura de sujeito mais íntegros em objetos menos íntegros. A revisão do modelo obrigatório BIBA de integridade não permite que os sujeitos de alta integridade leiam os objetos de baixa integridade. Isto pretende assegurar que a informação do sujeito de alta integridade, não seja corrompida pela baixa integridade do objeto. Entretanto no modelo da marca d´água baixa do sujeito é permitida a leitura de objetos de menor integridade por sujeitos mais íntegros, mas o resultado de tal leitura é o rebaixamento do nível de integridade

do sujeito ao nível do objeto lido [AMO94]. As características deste modelo são mostradas na Figura 2.3.

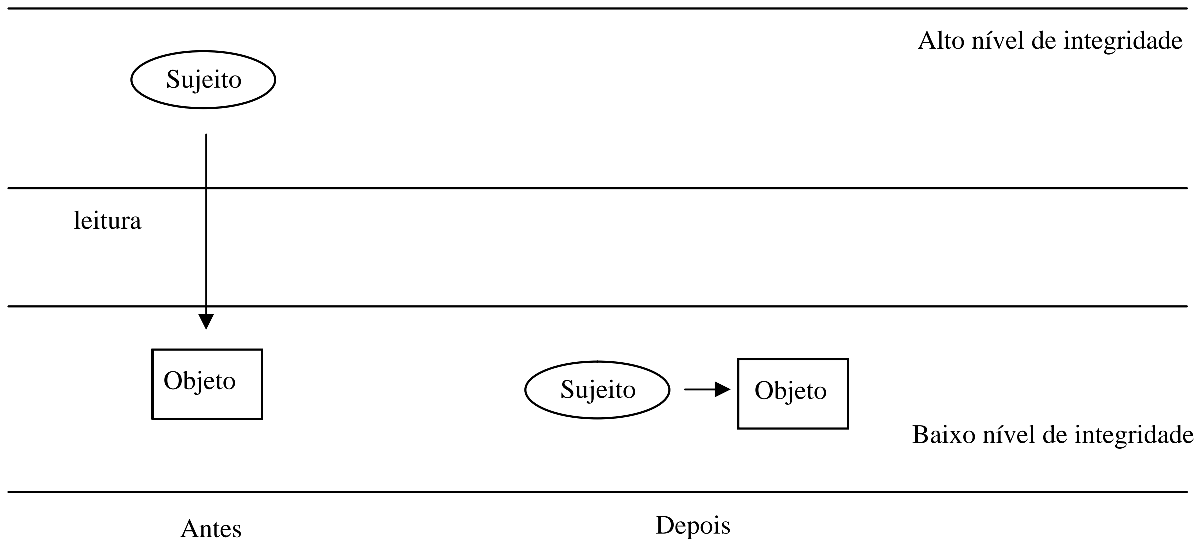

Figura 2.3. – Modelo BIBA da Marca D`Água Baixa do Sujeito

2.4.6. Modelo baseado em papéis

O modelo baseado em papéis (RBAC- Role-Based Access Control), tem seu ponto fundamental onde o usuário desempenha diferentes papéis em um sistema. Um papel pode ser definido como uma função em uma organização. Deste modo no modelo RBAC as permissões são atribuídas aos papéis e os usuários são autorizados a exercer papéis [SAN00]. As ações do modelo RBAC são descritas na Figura 2.4.

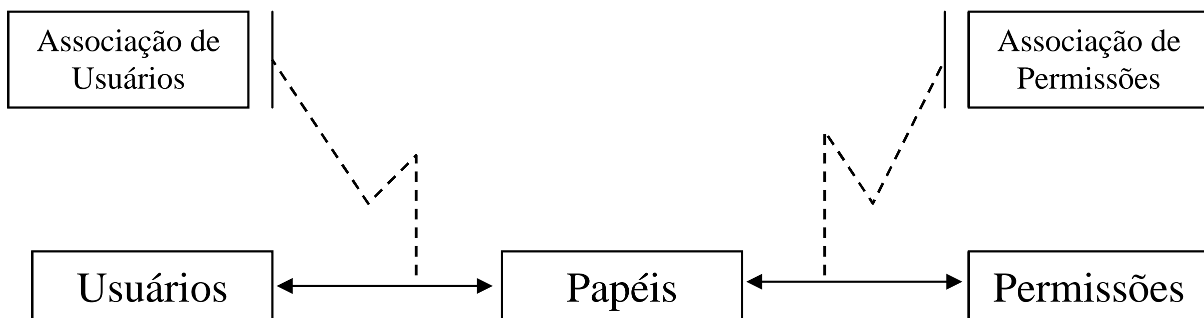

Figura 2.4. – Modelo RBAC Básico

A utilização do controle de acesso baseado em papéis gera facilidade na gerência de autorização, pois quando um usuário muda de atribuição, a manutenção das permissões dos papéis não sofre mudanças, já que o mesmo é desligado de um papel e assume outro.

O RBAC não é um conceito novo, mas só recentemente ganhou a atenção dos pesquisadores. Um modelo unificado denominado *RBAC-NIST* foi criado para tentar padronizar as várias tendências que têm surgido em modelos de papéis [SAN00]:

- **RBAC básico:** Implementa a infra-estrutura de base composta pelo usuário, permissões e papéis com suas respectivas semânticas. A relação entre os mesmos deve ser do tipo muitos-para-muitos. Este modelo define funções administrativas, interpretadas como suporte à revisão das associações usuário-papel – o que determina os usuários associados a um papel e vice-versa;
- **RBAC hierárquico:** Acrescenta ao RBAC básico uma hierarquia que permite estruturar papéis de maneira a refletir a hierarquia de responsabilidades reais de uma organização;
- **RBAC com restrições:** Está baseado no princípio do mínimo privilégio (suporta a *separação de tarefas*: o usuário só utiliza os direitos necessários para uma dada tarefa), impondo desta forma restrições na ativação de papéis conflitantes;
- **RBAC simétrico:** Inclui funções administrativas que dão suporte à revisão de associações permissão-papel – o que torna possível identificar as permissões associadas a um papel e os papéis que possuem determinadas permissões.

2.5. Mecanismos de segurança

Com a sofisticação dos sistemas computacionais e a abrangência nas suas aplicações, torna-se necessário prover uma crescente proteção de integridade. A proteção é a utilização de qualquer mecanismo para controlar o acesso de programas, processos ou usuários aos recursos definidos por um sistema computacional [AND72].

A função da proteção em um sistema computacional é fornecer algum mecanismo para aplicar políticas de segurança sobre a utilização de recursos. As políticas de segurança obedecem a critérios que foram explicados anteriormente. Um sistema de proteção deve ter a

flexibilidade de aplicar uma variedade de políticas que podem ser declaradas ao mesmo. A distinção entre mecanismo e política é muito importante:

- **Política:** Conjunto de regras;
- **Mecanismos:** Implementação da política.

No que se refere a controle de acesso esses mecanismos tomam o nome de monitor de referências, intervindo em vários níveis de um sistema. O monitor como responsável por intermediar todas as requisições de acesso aos objetos de um sistema deve ter algumas propriedades: deve ser inviolável, incontornável (sempre invocado) e pequeno o suficiente para permitir a verificação de sua correção.

2.5.1. Controle de acesso discricionário

Os mecanismos de controle de acesso discricionário (DAC) implementam políticas discricionárias, permitindo aos usuários atribuir direitos de acesso aos seus recursos computacionais de acordo com a sua necessidade; mas se o usuário não fizer uma atribuição correta dos recursos ou se o fizer permitindo acesso de cópia a outros sujeitos, o mesmo terá as suas informações disseminadas pelo sistema de forma não controlada. O controle de acesso discricionário não impõe nenhuma restrição à disseminação de direitos e a própria evolução da matriz de acesso. Devido à "facilidade" e flexibilidade para sua implementação, o controle de acesso discricionário é largamente utilizado nos sistemas atuais.

2.5.2. Controle de acesso obrigatório

Os mecanismos de controle de acesso obrigatório (MAC) implementam políticas obrigatórias, onde as regras de controle de acesso são impostas por uma autoridade central. Estes mecanismos, como visto em Bell e Lapadula e BIBA, implementam políticas multinível.

Os mecanismos MAC são mais difíceis de viabilizar que os mecanismos DAC, pois possuem regras mais rígidas e incontornáveis o que geram limitações à implementação deste modelo; deste modo os modelos conceituais necessitam de relaxamento de suas regras para poderem ser viabilizados em mecanismos comerciais.

2.5.3. Controle de acesso baseado em papéis

Os controles de acesso baseado em papéis (RBAC), regulam o acesso dos usuários aos recursos com base nas atividades desempenhada no sistema. Os papéis (roles) podem ser definidos como um conjunto de ações e responsabilidades associadas com uma atividade particular. A atribuição de direitos de acesso é destinada a papéis e não a usuários, e os usuários recebem permissão para assumir e fazer parte de um ou mais papéis.

2.6. Conclusão

Neste capítulo foram abordados os conceitos de segurança, onde se leva o entendimento que para se ter um sistema computacional seguro, este tem que atender a algumas propriedades básicas. Os modelos de segurança são formas de descrever as políticas de segurança, determinando o comportamento de entidades administradas por regras, as quais definem uma política de segurança.

Os modelos discricionário e obrigatório possuem diversas diferenças, onde o modelo obrigatório impõe regras mais rígidas e incontornáveis, mas são menos utilizados em sistemas comercias, devido ao fato de serem mais difíceis as suas implementações.

Capítulo 3

Proteção da Integridade do Sistema Operacional

Neste capítulo serão abordados os mecanismos que permitem manter a integridade *offline ou online*. Primeiramente serão mostradas as técnicas utilizadas pelos mecanismos para garantir a integridade. Para garantir a integridade uma série de controles podem ser implementados, inclusive envolvendo a autorização, pois a proteção contra mudança não autorizada depende de decisões de proteções a partir de mecanismos que visam a integridade. Os controles adicionais também serão abordados no decorrer deste capítulo.

3.1. Proteção de integridade

Um sistema computacional necessita manter seus dados íntegros; desta maneira tem que garantir que estes dados não sejam alterados por usuários que não tenham direitos para tal. Em nível de sistema operacional existem diversos mecanismos de proteção de integridade, estes mecanismos utilizam implementações de funções que realizam a verificação de integridade das informações do sistema. Algumas destas funções que também visam a verificação de integridade serão abordadas com maior detalhes nas sub-seções posteriores.

3.1.1. Funções *HASH*

As funções *hash* realizam um importante papel na criptografia moderna, fazendo parte dos mais diversos protocolos criptográficos existentes. Estas funções funcionam mapeando uma referência de dados de tamanho variável para uma seqüência de tamanho fixo de bits [BUR95].

O resultado do uso de uma função de *hash*, chamado de digital (*fingerprint*), *message digest* (MD) ou simplesmente *hash*, pode ser inicialmente comparado a um *checksum*, ou seja, ambos permitem a detecção de alterações dos dados utilizados para gerá-los, mas não fornecem qualquer recurso para a correção das mesmas [SCH96]. De forma mais abrangente, uma função de *hash* deve ter no mínimo as seguintes características:

- compressão: uma entrada x de tamanho arbitrário é mapeada para uma saída $h(x)$ de tamanho fixo;
- facilidade de cálculo: dada uma função h e uma entrada x, deve ser fácil calcular $h(x)$.

Considerando-se apenas estas duas características é possível classificar uma função de *checksum* XOR, por exemplo, como sendo uma função de *hash*. Mas apesar da aparente semelhança, as funções de *checksum* não dispõem das propriedades adicionais que fazem com que uma função de *hash* seja unidirecional. Essa diferenciação é facilmente justificada, uma vez que funções de *checksum* foram projetadas para detectar alterações causadas por interferências eletromagnéticas (ruídos existentes em canais de transmissão), e não alterações intencionais e maliciosas. Já as funções de *hash* utilizadas em criptografia foram especificamente projetadas para evitar a ocorrência de qualquer modificação que não possa ser detectada, seja esta intencional ou não [BUR95, MEN96].

Sendo assim, além da compressão e facilidade de cálculo, as funções de *hash* têm as seguintes propriedades adicionais:

- Unidirecionalidade: dado um *hash* de uma entrada não conhecida, $h(x)$, deve ser muito difícil obter a entrada x em parte ou na íntegra; essa dificuldade deve manter-se mesmo que parte de x seja conhecida;
- Resistência a colisões fracas: dado uma determinada entrada x, deve ser muito difícil encontrar outra entrada x', tal que $h(x') = h(x)$;
- Resistência a colisões fortes: deve ser muito difícil encontrar duas entradas quaisquer, x e x', tal que $h(x') = h(x)$; a principal diferença desta em relação às duas anteriores é que o atacante tem liberdade total para escolher tanto x quanto x';

- Resistência a colisões próximas: deve ser difícil encontrar duas entradas *x* e *x'* tal que *h(x)* e *h(x')* difiram em apenas um número pequeno de bits;
- Inexistência de correlação: os bits da entrada *x* e da saída *h(x)* não devem ser correlacionados; a idéia aqui é que cada bit da entrada afete todos os bits da saída; isto é conhecido como efeito avalanche.

Todos esses aspectos comentados tornam as funções de *hash* ferramentas extremamente úteis para o controle de integridade de arquivos, permitindo a criação de uma representação compacta e única de uma determinada massa de dados.

3.1.1.1. Funções de hash customizadas

Estas funções foram projetadas desde o seu início para o propósito de geração de *hash*. Sendo assim, elas não sofrem das limitações impostas pelo uso de mecanismos já existentes, como a cifragem de blocos e a aritmética modular, tornando-as também independentes da existência desses subcomponentes para sua aplicação.

Entre os diversos algoritmos existentes, o Message Digest 4 (MD4) é considerado como um marco histórico, pois é através do MD4 que são baseados os algoritmos de hash mais populares, que compõem praticamente o padrão em relação à eficiência e segurança para a geração de hashes. Os algoritmos mais conhecidos atualmente são o MD5 (Message Digest 5) e SHA1 (Secure Hash Algorithm), os quais serão explicados nos próximos itens.

3.1.1.2. Message Digest 4 (MD4)

O número 4 em uma série de algoritmos criados por Ronald Rivest, no MIT, o MD4 foi projetado especificamente para ser extremamente eficiente em máquinas com arquitetura de 32 bits, gerando *hash* de 128 bits. O MD4 [RIV92] opera sobre blocos de 512 bits, os quais são subdivididos em 16 blocos de 32 bits, resultando daí um conjunto de 4 blocos de 32 bits cada, que, concatenados, formam o *hash* de 128 bits. As operações utilizadas internamente compreendem soma, deslocamento, XOR e operações lógicas OR e AND. Após sua publicação, alguns pesquisadores conseguiram atacar algumas das etapas da operação do MD4, mas não o algoritmo como um todo. Apesar destes ataques não poderem ser estendidos para o algoritmo inteiro, Rivest realizou em seguida algumas modificações no algoritmo de

forma a torná-lo mais seguro, resultando na criação do MD5, o qual opera com blocos de 512 bits, subdivididos em 16 blocos de 32 bits cada [RIV02].

As diferenças entre MD5 e seu antecessor MD4 são as seguintes:

- Foi adicionada uma quarta etapa; no MD4 eram somente 3;
- Cada passo tem uma constante aditiva única (*t*); no MD4 as constantes são reutilizadas;
- A função da etapa dois *G* = ((*x* and *y*) or (*x* and *z*) or (*y* and *z*)) foi trocada por G = (*x* and *z*) *or* (*y* and not(*z*)); o motivo era torná-la menos simétrica;
- Cada passo adiciona o resultado do passo anterior, promovendo um efeito avalanche mais rápido;
- A ordem em que os blocos de 32 bits *Mi* eram acessados nas etapas 2 e 3 foi trocada para fazer com que ficassem menos parecidas;
- Os valores de deslocamento *s* foram otimizados para causar um efeito avalanche melhor; esses valores são diferentes em cada etapa.

Apesar de melhorias em relação ao MD4, alguns estudos descritos em [BOE93], apontaram a existência de fraquezas no algoritmo que permitiram a geração de colisões em algumas situações especiais (restritas). Felizmente, tal ataque, na prática, não representa qualquer problema para o MD5, nem mesmo serve para afirmar que sua propriedade de resistência a colisões tenha sido violada. [SCH96] De qualquer forma, o MD5 é um dos algoritmos mais difundidos, estando presente na grande maioria das aplicações que utilizam criptografia para obter algum tipo de segurança, como, por exemplo, *Secure Sockets Layer* (SSL), *Secure Shell* (SSH) e *Pretty Good Privacy* (PGP).

3.1.1.3. Secure Hash Algorithm (SHA1)

O SHA1 foi desenvolvido pelo NIST e pela NSA para ser utilizado como o algoritmo de hash padrão para algumas aplicações do governo dos Estados Unidos [BUR95, HAN01].

Também baseado no MD4, opera com blocos de 512 bits, subdivididos em 16 blocos de 32 bits cada. Seu algoritmo é por conseqüência bastante parecido com o do MD5, com o pré-processamento (*padding* e *length block*) praticamente idênticos.

Segue abaixo uma comparação das melhorias do MD5 em relação ao MD4 com as modificações existentes no SHA1 [SCH96]:

- SHA1 também foi adicionada uma quarta etapa, mas a segunda e quarta rodadas utilizam a mesma função *f*;
- SHA1 manteve o esquema do MD4 no caso da constante *t*, reutilizando as constantes a cada 20 rodadas;
- SHA1 não modificou a função *G*, mantendo a versão do MD4;
- "Cada passo adiciona o resultado do passo anterior, promovendo um efeito avalanche mais rápido"; o mesmo foi feito no SHA1, mas uma quinta variável *e* foi adicionada, fazendo com que os ataques ao MD5 descritos em [BOE93], não possam ser aplicados ao SHA1;
- SHA1 implementa o acesso aos blocos de entrada de uma forma completamente diferente do MD4 e do MD5, utilizando um código cíclico de correção de erros;
- Os valores de deslocamento do MD4 foram mantidos no SHA1.

A diferença mais visível do SHA1 em relação ao outros algoritmos é o tamanho do hash gerado, que é de 160 bits ao invés dos 128 bits do MD5 e do MD4. Isso faz com que o SHA seja mais resistente a ataques, e talvez como conseqüência disso, não há ataques criptográficos conhecidos que tenham sido aplicados com sucesso contra o mesmo[SCH96]. Mais detalhes sobre o algoritmo e sua implementação pode ser obtido em [BUR95].

3.2. Sistemas clássicos

Em muitos sistemas operacionais, a segurança está baseada no controle de acesso aos recursos, basicamente operações de leitura e escrita. Por exemplo, o controle de acesso a arquivos no Unix é realizado com mecanismos de permissões (leitura, escrita e execução). Uma aplicação de *log*, mesmo precisando escrever ao final do arquivo de *log*, poderá modificar qualquer parte do arquivo de *log*, o que pode ser utilizado por um intruso que obteve acesso para apagar os rastros de uma invasão, ocultando sua ação. Entretanto, o excesso de granularidade pode causar alto impacto no desempenho do sistema operacional, tendo em vista que cada pedido realizado será analisado para verificar se o processo envolvido tem a devida permissão.

Os mecanismos de controle de acesso mais utilizados nos sistemas operacionais atuais são: listas de controle de acesso e capabilities, os quais serão melhores explicados a seguir.

3.2.1. Listas de controle de acesso ACL

Uma lista de controle de acesso (*Access Control List - ACL*) armazena os direitos de acesso dos sujeitos a objetos, separadamente para cada objeto. A ACL corresponde a uma coluna da matriz de acesso e estabelece os sujeitos que podem acessar um certo objeto [AMO94, TAN03]. Na Figura 3.1, existem três processos (A, B e C) e três arquivos (F1, F2 e F3). Cada arquivo apresenta uma ACL associada ao mesmo. O arquivo F1 tem duas entradas em sua ACL. A primeira entrada mostra que qualquer processo em A pode ler e escrever no arquivo. A segunda entrada mostra que qualquer processo em B é capaz de ler o arquivo. Todos os outros acessos desses usuários e todos os acessos de outros usuários são proibidos. Os direitos são outorgados pelo usuário, e não pelo processo. No que se diz respeito ao sistema de proteção, qualquer processo em A pode ler e escrever no arquivo F1. Não importa se há mais que um processo. O fator importante é o proprietário e não o ID do processo.

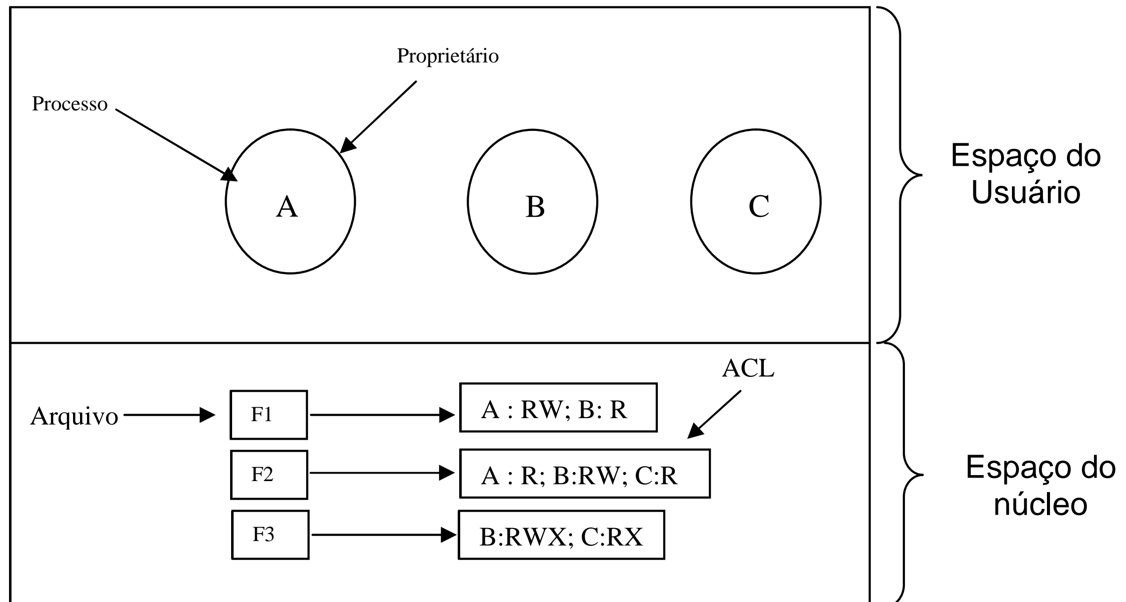

Figura 3.1. – Modelo da Lista de Controle de Acesso

O arquivo F2 tem três entradas em sua ACL: A, B e C podem ler o arquivo e, além disso, B também pode escrever no mesmo. Nenhum outro acesso é permitido. O arquivo F3 é aparentemente um programa executável, pois B e C podem ler e executá-lo e B pode escrever no mesmo.

Diversos Sistemas Operacionais suportam o conceito de grupo de usuários. Os grupos têm nomes e podem ser incluídos na *ACLs*. Duas variações são possíveis na semântica de grupos. Em alguns sistemas, cada processo tem um identificador de usuário (*UID*) e um identificador de grupo (*GID*). Verificando a condição de se trabalhar com grupo, quando o acesso a um objeto é requisitado, é feita uma verificação usando o *UID* e o *GID* de quem requisitou. Se os mesmos estiverem presentes na *ACL*, os direitos apresentados serão concedidos. Se a combinação (*UID,GID*) não estiver na lista, o acesso não será permitido.

Um problema associado ao uso de ACLs em sistemas com muitos usuários é que a lista pode ser muito grande, fazendo-se necessário utilizar abreviações na lista. Definições padrões para usuários não presentes na lista, grupos de usuário e caracteres especiais são alguns dos recursos utilizados para abreviar a lista.

3.2.2. Capabilities

Uma outra forma de implementar o modelo de matriz é percorrer a matriz pelas linhas. Para realizar isto, é associada a cada processo uma lista de objetos aos quais se pode ter acesso, juntamente com uma indicação de quais operações são permitidas. A lista utilizada é denominada, lista de capacidades (capabilities) ou C-list [GEH82, TAN03].

Cada capacidade garante ao portador certos direitos sobre um certo objeto. Na Figura 3.2, o processo pertencente ao usuário A pode ler os arquivos F1 e F2, por exemplo. Normalmente, uma capacidade consiste em um identificador de arquivo e um mapa de bits para os vários direitos.

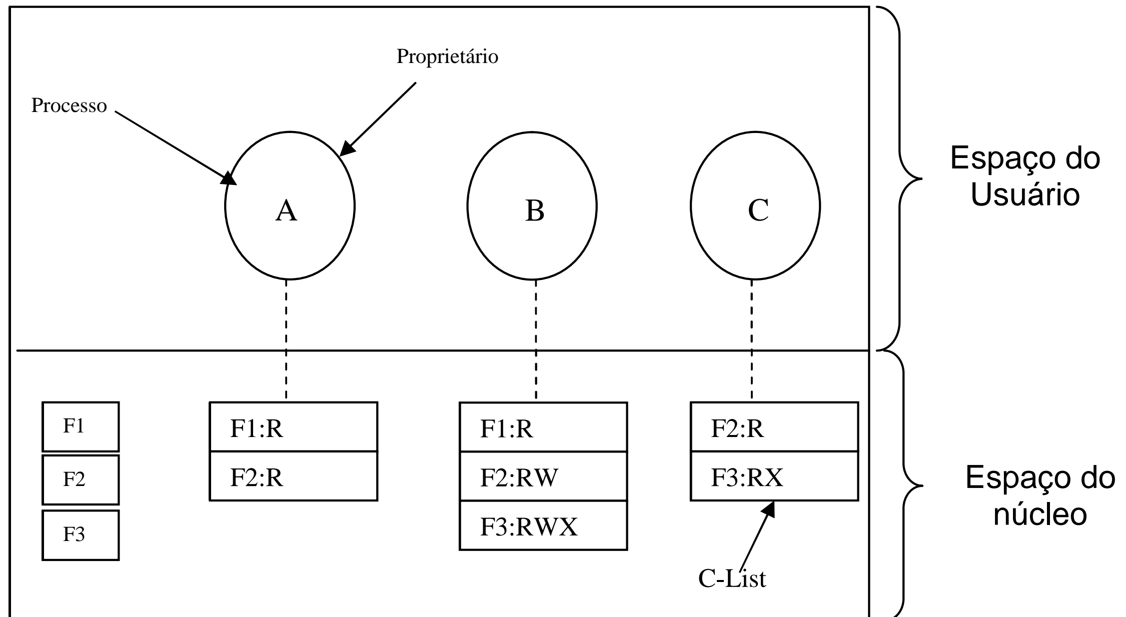

Figura 3.2. – Modelo de Capabilities

As listas de capacidade são objetos propriamente ditos e podem ser apontadas e apontarem outras listas de capacidade.

A lista de capacidades é mais interessante em termos de implementação que a de ACL. Em sistemas que implementam ACL, tanto a lista de controle de acesso quanto a identidade do processo estão sob o controle do sistema operacional, de tal forma que o usuário só pode manipulá-las por intermédio de uma API.

3.3. Proteção de integridade *batch* (*offline*)

Os sistemas de proteção de integridade *batch* são ferramentas utilizadas para monitoramento de arquivos através de agendamentos ou intervenções do administrador do sistema, não ocorrendo desta forma uma verificação dinâmica. Diversas dessas ferramentas são utilizadas com sua implementação realizada através de algoritmos de *hash*, para se ter uma verificação de integridade mais confiável.

3.3.1. *TripWire*

O *Tripwire* é uma ferramenta que visa realizar o monitoramento de arquivos de um sistema computacional, com o intuito de garantir a integridade dos arquivos monitorados. O *Tripwire* teve a sua primeira implementação em 1992, resultante de estudos realizados no

Departamento de Ciências da Computação da Universidade de Purdue, Estados Unidos. Desde então, sua estrutura tem servido de modelo para vários outros projetos [KIM95].

O *Tripwire* não pode ser considerado um IDS (*Intrusion Detection Service*), pois o mesmo não tem capacidade de ser configurado para realizar ações contra intrusos que desejam afetar os arquivos que o mesmo está monitorando. O monitoramento realizado pelo *Tripwire* não é dinâmico, pois a verificação da integridade dos arquivos deverá ser feita com intervenção do administrador ou programada. A execução programada não é muito aconselhada, pois a verificação deve ser preferencialmente realizada pelo administrador. Desta forma, tem uma execução com maior segurança, inclusive indicando-se o caminho completo para os executáveis do sistema.

Quando o *Tripwire* é instalado deve ser criado um banco de dados com informações dos arquivos que serão monitorados. Desta forma, poderão ser feitas checagens periódicas visando identificar qualquer alteração ocorrida nestes arquivos, usando-se como base as informações armazenadas no banco de dados criado. O banco de dados com as informações dos arquivos monitorados é armazenado de forma criptografada, visando garantir que não possa ser alterado sem que seja usada uma chave criptográfica.

Entre os algoritmos de *hash* disponibilizados no *Tripwire*, têm-se: *Message Digest* 2 (*MD2), Message Digest 4 (MD4), Message Digest 5 (MD5), Snefru*, HAVAL-128bits, *Secure Hash Algorithm (SHA1)*, CRC-16 e CRC-32. Uma característica interessante do *Tripwire* é a possibilidade de utilização de algoritmos diferentes para cada arquivo, ou conjunto de arquivos, permitindo que o administrador do sistema escolha o nível de segurança para os arquivos que deseja proteger. Pode-se então, por exemplo, utilizar o CRC-32, que é extremamente rápido, para arquivos de menor importância e o *SHA1* para arquivos críticos do sistema [KIM95]. Nas primeiras implementações, o snapshot gerado pelo *Tripwire* não tinha nenhum mecanismo de proteção, tornando-se um ponto extremamente vulnerável. Este problema foi resolvido nas implementações mais recentes com o uso de assinaturas digitais.

O *Tripwire* é configurável, não muito diferente de outras ferramentas para Linux, podendo ser definidas várias regras para verificação de integridade de arquivos, de acordo com as necessidades de cada administrador. Nesta definição, além de quais arquivos serão

verificados, pode-se definir que atributos do arquivo serão verificados, como por exemplo, proprietário, grupo, tamanho, permissões, *checksum,* etc.

O uso do *Tripwire* pode trazer uma melhoria no nível de segurança, tendo em vista que a constante verificação da integridade dos arquivos, apesar de não bloquear possíveis ataques de forma dinâmica, pode determinar a dimensão dos danos causados por estes ataques e dependendo da freqüência com que é realizada, pode evitar maiores estragos pela identificação de pontos ultrapassados no sistema de segurança.

O *Tripwire* originalmente era um software proprietário, mas a partir de outubro de 2000, na sua versão 2.2.1 para o sistema operacional Linux teve seu código licenciado sob os termos da GPL (General Public License) sendo, portanto, um software livre.

3.3.2. Advanced Intrusion Detection Environment (AIDE)

O AIDE é uma iniciativa isolada para a criação de um utilitário livre que substitua o *Tripwire*. Os seus autores são: Rami Lehti e Pablo Virolainen. Segundo os próprios autores, em [LET05], o AIDE implementa funcionalidades adicionais às implementadas na versão livre do *Tripwire.* O AIDE funciona de maneira similar ao *Tripwire*, utilizando os mesmos algoritmos de *hash*, porém não são implementados quaisquer tipos de proteção ao snapshot e nem mesmo assinaturas digitais.

3.4. Sistemas adicionais de proteção

Os sistemas adicionais, ao contrario dos clássicos, são sistemas que são desenvolvidos por grupos de pesquisa de fora das empresas geradoras dos sistemas operacionais. Desta forma, estes sistemas são adicionados aos sistemas operacionais através de pacotes de segurança ou patches de atualização de alguns outros tipos de sistemas.

3.4.1. Linux Intrusion Detection System (LIDS)

O LIDS, apesar do nome, é um *patch* para o kernel do sistema operacional Linux que implementa uma série de controles adicionais sobre os recursos do sistema. Entre esses controles estão [TAM05]:

- **Proteção de arquivos:** arquivos protegidos pelo LIDS não podem ser modificados por nenhum usuário, inclusive o administrador; arquivos podem ser escondidos do restante do sistema;
- **Proteção de processos:** processos podem ser protegidos inclusive da ação do administrador, e, assim como os arquivos, podem ser escondidos;
- Controle granular sobre o acesso a recursos do sistema através de *Access Control Lists* (ACLs);
- Bloqueio de acesso a vários recursos do sistema como portas TCP/IP, acesso direto a dispositivos (e.g. */dev/hda*), entre vários outros.

Embora não implemente nenhum controle de integridade de arquivos, o LIDS possui características bastante interessantes para a garantia da segurança do *snapshot*, uma vez que permite a limitação dos poderes do super-usuário, resolvendo alguns problemas de segurança.

3.4.2. Linux Security Modules (LSM)

A estrutura do LSM tem o objetivo de permitir a utilização de diferentes modelos de segurança para o Linux minimizando o impacto ao kernel. A generalidade do LSM permite que controles de acesso sejam executados de forma eficaz sem necessitar de um *patch* para o kernel [WRI02].

A interface de chamada do sistema fornece uma abstração para o espaço do usuário interagir com o kernel, e uma posição temporária para intermediar o acesso. Na realidade, nenhuma modificação do kernel é necessária para sobrescrever entradas na tabela de chamadas do sistema, trabalhando de forma normal para intermediar esta interface, através de módulos do kernel.

Uma abstração básica da interface do LSM deve intermediar o acesso aos objetos internos do kernel. A arquitetura do LSM hooks pode ser melhor visualizada na Figura 3.3,

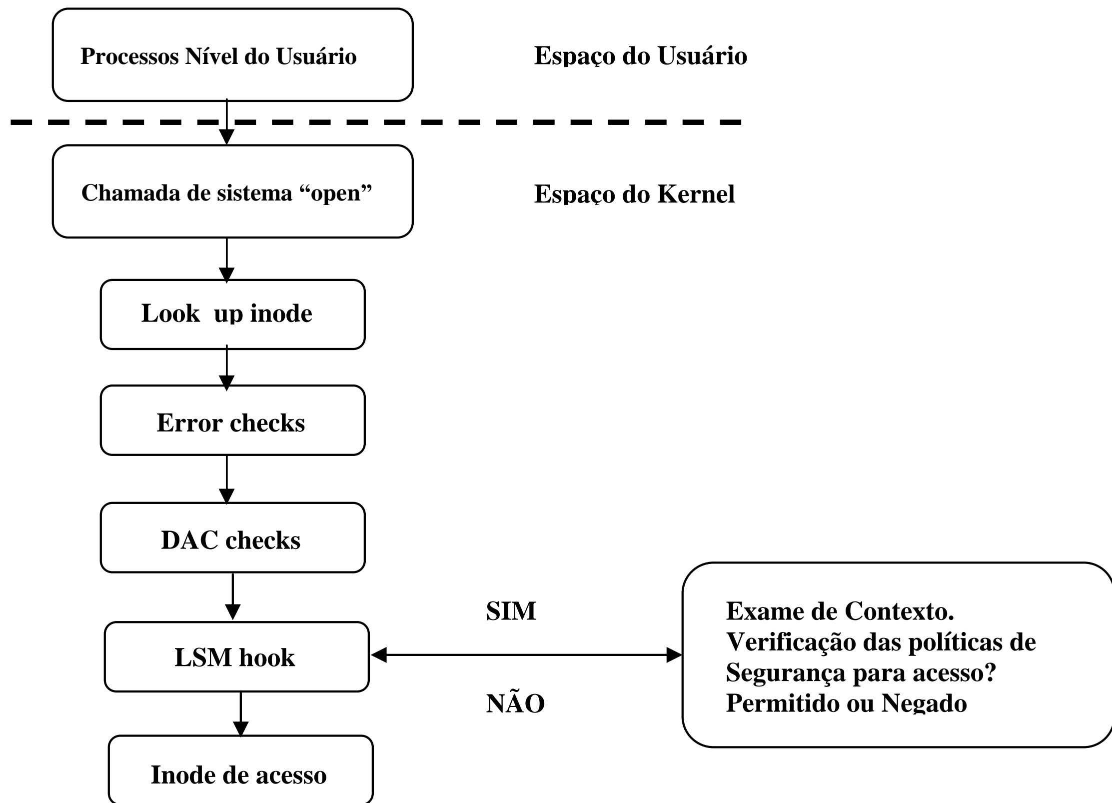

Figura 3.3. – Arquitetura LSM Hooks

O LSM permite que os módulos façam a intermediação do acesso aos objetos do kernel colocando os hooks no código do kernel, como mostrado na Figura 3.4. Antes que o kernel acesse um objeto interno, o hook criado realiza uma chamada para função que o módulo de LSM irá fornecer. Este módulo deixa o acesso ocorrer, ou nega o acesso, forçando um código de erro de retorno.

A estrutura do LSM força os mecanismos existentes no kernel a traduzir os dados fornecidos pelos usuários. A estrutura do LSM também permite o intermédio direto ao acesso às estruturas de dados do núcleo do kernel [WRI02]. Com tal abordagem, a estrutura do LSM tem acesso ao contexto completo do kernel imediatamente antes que o kernel execute o serviço solicitado. Isto melhora a granularidade no controle de acesso.

O projeto do LSM escolheu limitar o seu espaço para suportar a funcionalidade do controle de acesso do núcleo requerida pelos projetos de segurança existentes no Linux. Esta limitação permitiu que a estrutura do LSM permaneça conceitualmente simples.

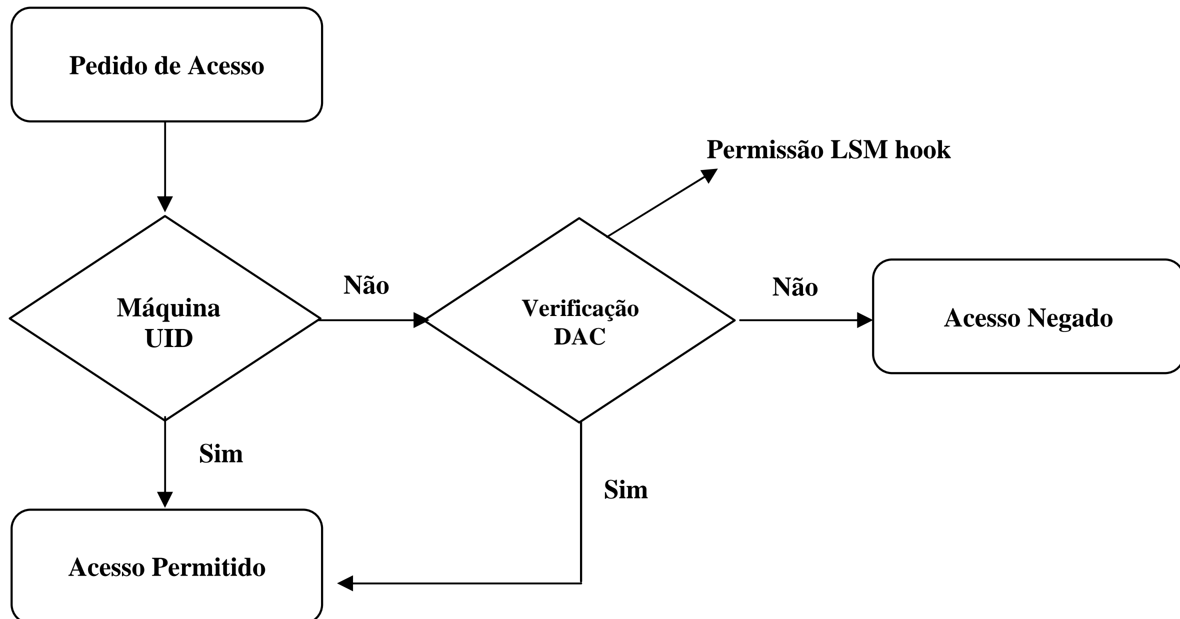

Figura 3.4. – LSM hook de permissão

3.4.3. Verificação de integridade de arquivos pelo kernel do sistema operacional

A técnica proposta para realizar a verificação de integridade, passa para o kernel do sistema operacional a responsabilidade de verificar a integridade de todo código executável ou arquivo aberto. Deste modo, visa detectar modificações antes que estas possam ter qualquer efeito sobre o sistema, evitando que partes do sistema comprometidas por um invasor possam ser ativadas. Ao lançar um novo processo, carregar bibliotecas dinâmicas ou abrir um arquivo, o kernel efetua uma verificação necessária aprovando ou bloqueando sua execução ou acesso e tomando as medidas necessárias em caso de problemas [BOR01].

O funcionamento da verificação de integridade de arquivos da técnica proposta é composto por duas partes:

- **Parte estática:** Formada basicamente por uma lista de arquivos a monitorar;
- **Parte dinâmica:** Formada pelos mecanismos para realizar a monitoração dos arquivos contidos na lista da parte estática.

A lista de monitoração deve ser construída a partir do sistema em um estado inicial íntegro, ou seja, com o sistema operacional e demais aplicativos sendo instalados a partir de fontes confiáveis, portanto sem a presença de código intruso como *backdoors* e *trojans*. Sobre

esse estado inicial confiável é construída uma base de assinaturas, formada pelos nomes de caminho dos arquivos associados a informações que permitam a detecção de violações.

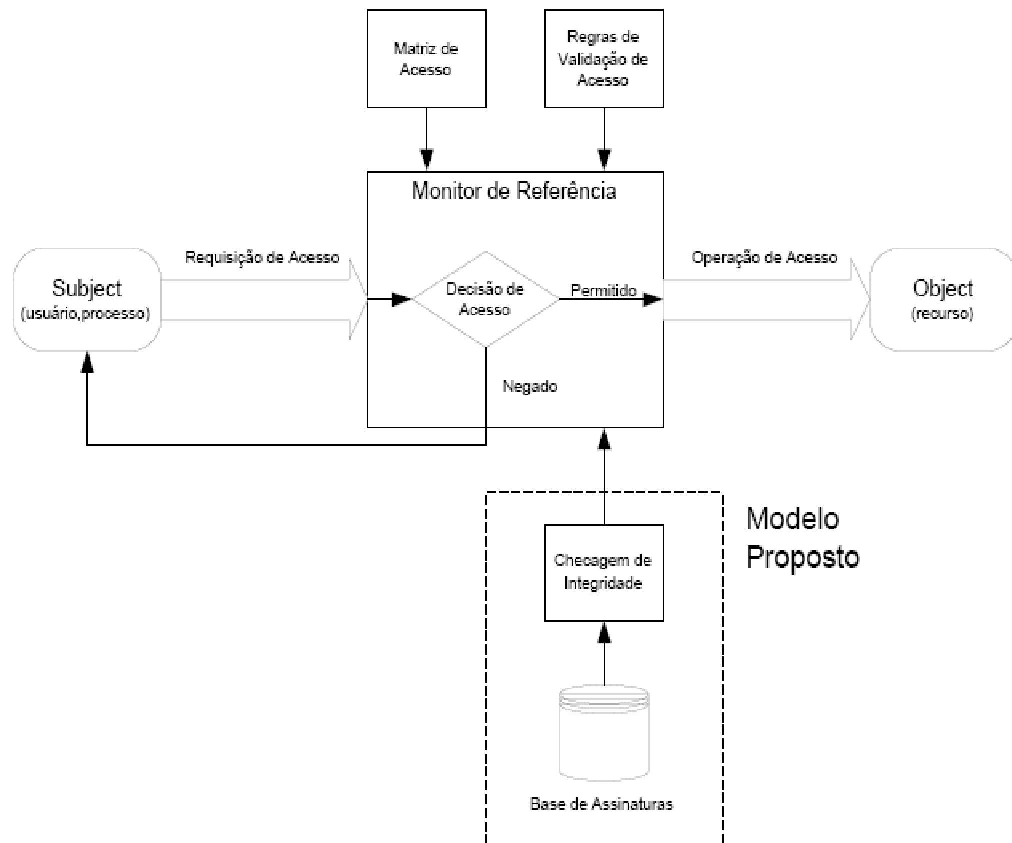

Figura 3.5. – Representação do Monitor de Referência com a incorporação do Modelo Proposto [BOR01]

Uma das maneiras de se representar um modelo de controle de acesso é através de um monitor de referência. Independentemente do modelo utilizado, o núcleo do monitor não se altera, seja para um modelo discricionário, obrigatório, baseado em regras ou outro. Esta técnica está baseada no modelo discricionário típico de um sistema UNIX. A Figura 3.5 demonstra como é o funcionamento do modelo sugerido, sem alterações significativas do modelo básico.

As alterações necessárias no kernel para sistemas operacionais respeitando o padrão POSIX, seriam duas. A primeira é a interceptação da chamada do sistema *execve*, responsável pela execução de novos processos. A segunda é a interceptação da chamada do sistema *open*,

responsável pela abertura de arquivos pelos processos. A verificação de integridade pode ser realizada comparando o arquivo atual com uma cópia íntegra previamente armazenada. Como essa situação é claramente inviável, pode-se fazer o uso de um *message digest* do arquivo em questão. Além disso, podem-se também armazenar outros atributos do arquivo, como direitos de acesso, proprietário, grupo e datas. Para evitar que o invasor possa modificar o *message digest* de um arquivo que o mesmo tenha adulterado é utilizada uma assinatura digital. Com o uso de algoritmos de chave assimétrica, cada *message digest* é criptografado com a chave privada definida para o sistema, formando uma assinatura digital. Com isso forma-se uma base de assinaturas dos arquivos, composta por tuplas formadas pela assinatura e pelos atributos. Somente a chave pública e esta base de assinaturas são utilizadas pela rotina incorporada ao kernel para verificação de integridade.

A técnica proposta do ponto de vista conceitual é muito simples, mas existem alguns problemas a serem resolvidos para que se possa aplicá-la de forma viável. Os principais problemas são:

- **Localização da chave pública:** Se a chave pública puder ser trocada pelo invasor, este poderá recriar a base de assinaturas com seu par de chaves e o sistema estará comprometido;
- **Localização da base de assinaturas:** Essa estrutura é menos sensível que a chave pública com relação à integridade, mas é muito importante para a operação do sistema. Se ela for adulterada, o arquivo associado à entrada alterada ficará inacessível. Se ela for apagada, o sistema ficará inoperante. Portanto, sua proteção é essencial;
- **Arquivos não assinados e em diretórios de usuários:** A técnica proposta trata apenas dos arquivos assinados, não prevendo nenhuma política com relação aos demais;
- **Desempenho:** Os algoritmos de *hash* e criptografia são computacionalmente pesados e podem levar a problemas de desempenho;
- **Administração:** Um sistema de informação está em constante evolução. Instalações de novos softwares ou atualizações são necessárias freqüentemente, o que pode representar um problema para a manutenção da base de assinaturas.

3.5. Conclusão

Devido à necessidade do sistema operacional em manter seus dados íntegros, diversos mecanismos foram propostos para atender a propriedade de segurança referente à integridade. Deste modo, os sistemas operacionais atuais possuem aplicações referente a proteção de integridade incorporadas em sua construção e muitas outras aplicações são comercializadas com intuito de manter um sistema íntegro, mas infelizmente não totalmente seguro. As soluções que já vem adicionada aos sistemas operacionais são conhecidas como sistemas clássicos. As outras soluções comercializadas no mercado, ou por empresas especializadas em segurança ou por grupos de pesquisa que participam de projetos de software livre, normalmente são pacotes de soluções de segurança para serem instaladas no sistema operacional ou são *patches* de atualização dos sistemas e são conhecidas como sistemas adicionais.

Capítulo 4

Mecanismos de Controle de Acesso Obrigatório

Os mecanismos de controle de acesso obrigatórios mais comuns implementam o modelo obrigatório de Bell-Lapadula ou BIBA, utilizando políticas multí-nível com base em *security-labels* [AMO94].

O modelo de confidencialidade Bell-Lapadula (BLP), foi proposto para especificar formalmente o controle de acesso em aplicações militares. Em tais aplicações, os sujeitos e os objetos são estratificados em níveis diferentes de segurança. Num mesmo nível são consideradas subdivisões/compartimentos, denominados categorias. Bell e Lapadula especificaram um conjunto de regras formais para representar as relações de dominância entre níveis e categorias, de tal modo que a confidencialidade seja preservada no sistema [LAN81].

O modelo de integridade BIBA é comumente referenciado como o dual do modelo Bell-Lapadula, pois se baseia nos mesmos princípios para especificar regras que visam à integridade dentro de um sistema de aplicações militares [AMO94, BIB77, LAN81].

Nas seções a seguir, serão abordadas as principais características de alguns mecanismos de controle de acesso implementando políticas multi-nível com base em *security-label* visando a confidencialidade e a integridade.

4.1. Modelo do cofre

Este modelo propõe uma abordagem para problemas como a inflexibilidade dos modelos de segurança. O modelo do cofre envolve a incorporação criptográfica dentro de uma

infra-estrutura de segurança para o kernel do sistema operacional. Os componentes desta infra-estrutura podem ser separados em dois tipos: partes de repositórios (cofres) e mecanismos de proteção [PAY04].

Um “cofre” é simplesmente uma estrutura de dados que confina dados sensíveis, separando os privilégios dos usuários de suas identidades; o kernel de segurança controla com cuidado o acesso a estes dados de acordo com um pequeno conjunto simples de regras, predefinidas. Enquanto a segurança da maioria dos dados confinados dentro de um cofre é relacionada como confidencial, a estrutura do cofre também fornece a proteção de integridade com a prevenção de modificação desautorizada e assegura a disponibilidade dos dados às partes autorizadas. Há cinco tipos diferentes de cofres em uso dentro do modelo, cada um com uma finalidade diferente e acessível por diferentes maneiras.

4.1.1. Cofre de usuário

Cada usuário no sistema tem seu próprio cofre que é acessível somente a cada usuário correspondente. Conseqüentemente o conteúdo do cofre de um usuário determina seus privilégios no sistema. O usuário pode armazenar virtualmente qualquer tipo de valor secreto dentro de seu cofre e recuperá-lo quando necessário. No geral há três tipos de objetos armazenados dentro do cofre de um usuário: chaves, bilhetes e impressões digitais. As chaves são os valores secretos que estão disponíveis para serem utilizados pelas aplicações selecionadas que as solicitarem quando utilizadas pelo usuário. O cofre de um usuário está armazenado e criptografado entre sessões e é decifrado quando o usuário entra requisitando o acesso ao seu conteúdo para a duração dessa sessão [PAY04].

4.1.2. Cofre global

Existem dois tipos de cofres globais. O cofre ou o GPRIV (Cofre Global Privado) confidencial global são equivalente a cofres de usuário. Somente o kernel do sistema pode acessar diretamente o GPRIV que age como um repositório central para os valores sensíveis requisitados pelo kernel, executa determinadas tarefas, mas onde o valor próprio deve ser protegido de todos os usuários. A maioria dos artigos define GPRIV como chaves de proteção de arquivos, que serão descritas em detalhe mais adiante. O GPUB (Cofre Global Público) é o oposto do GPRIV, o mesmo confina os valores que devem ser acessíveis por todos os

usuários no sistema. Entretanto, os usuários não podem modificar os valores confinados no GPUB. Assim GPUB cumpre o papel de assegurar-se de que determinadas partes dos dados estejam sempre disponíveis e com sua integridade garantida.

4.1.3. Cofre escrow (Garantia)

O cofre *escrow* comporta-se similarmente ao cofre GPRIV. Entretanto, o cofre escrow existe somente para confinar as chaves para os objetos protegidos. Desta forma, as chaves podem ser recuperadas se requisitada pelo administrador de segurança, a fim de contornar as proteções do arquivo.

4.1.4. Cofre fundamental

O cofre fundamental confina as chaves para cifragem dos outros quatro cofres. Deste modo fornece uma camada adicional de abstração, permitindo que uma única chave seja usada para proteger a infra-estrutura. O administrador do sistema pode escolher o mecanismo mais apropriado para armazenar e proteger esta chave, de acordo com seus requisitos de segurança.

4.1.5. Proteção de arquivos

A proteção de leitura e escrita fazendo a utilização de cofres faz-se com que todos os tickets de acesso dos arquivos sejam mantidos em um único lugar, nos cofres de usuário. Deste modo, aumenta a flexibilidade aumentada mantendo todos os arquivos em um único diretório cifrado, com algumas execuções. Alguns diretórios podem conter uma mistura de arquivos textos e de arquivos cifrados, portanto, se necessário, cada arquivo protegido pode utilizar uma chave diferente [PAY04].

Porém, para se ter um sistema de proteção de arquivos completo, é necessário fornecer uma proteção não somente de integridade (escrita), mas também de confidencialidade e de cofres, permitindo que as chaves de proteção de arquivos sejam usadas para ambos os formulários de proteção.

A proteção de *tickets* e arquivos compartilhados é um outro ponto deste esquema de proteção. Os usuários também necessitam conceder controle a outros usuários para que os

mesmos possam revogar um acesso a objetos e isto é realizado atribuindo um símbolo (*ticket*) ao usuário. Quando o proprietário de um objeto decide proteger este objeto, é gerado um *ticket* para o objeto e a chave usada para obter a proteção, é armazenada no cofre global privado (GPRIV).

4.1.6. Vantagens dos cofres

A arquitetura de cofres oferece algumas vantagens para projetos de sistemas confiados, implementando uma forte segurança, assim a característica da segurança dos cofres é fornecer um sistema de segurança similar às características do modelo obrigatório de segurança.

A flexibilidade alcançada com o modelo do cofre é muito significativa. Pois, não somente o administrador de segurança pode configurar um sistema para reforçar a política apropriada de segurança usando fortes mecanismos de proteção, como mesmos os usuários também podem fazer uso das vantagens destas características de segurança para executar suas próprias requisições de segurança quando necessário [PAY04].

4.2. Modelo de autorização multi-nível

Neste esquema de autorização é proposta a implementação de dois níveis de proteção. A proteção de objetos persistentes (sem granularidade) do sistema é realizada por um servidor central de autorização. Este servidor de autorização pode gerenciar os direitos de acesso para operações de nível mais elevado no. Um segundo nível de proteção é implementado para cada local do sistema: um kernel de segurança é responsável pela verificação de todos os acessos aos objetos locais e, além disso, é responsável pelo controle local dos direitos de acesso para todos os objetos locais não-persistentes [NIC97].

4.2.1. Servidor de autorização

O servidor de autorização é responsável pelo controle dos direitos de acesso para objetos persistentes do sistema. Este servidor decide cada acesso ao objeto persistente no sistema, se este acesso deve ser autorizado ou negado. A fim de examinar estas decisões, as autorizações dos usuários são armazenadas em uma matriz de acesso (nas quais são definidos

os direitos para acessar objetos persistentes) e gerenciam as regras que permitem *capabilities* para ser construída e entregue aos objetos que são autorizados a acessar os objetos persistentes.

O servidor de autorização pode ser projetado tolerante a faltas e tolerante a intrusão. Onde várias técnicas podem ser utilizadas; para utilizar a técnica de redundância de fragmentação, o servidor de autorização tem que ser composto de diversos locais da segurança, e somente alguns destes locais de segurança são confiáveis. Isto significa que a falha ou a intrusão, em uma minoria dos locais de segurança, não compromete a segurança do sistema inteiro.

4.2.2. Segurança do kernel

Cada local do sistema tem um kernel de segurança. Este kernel de segurança possui dois papéis principais:

- Verifica todos os acessos aos objetos locais, se persistentes ou não-persistentes;
- Controle autônomo de todos os direitos de acesso para objetos não-persistentes locais.

O kernel de segurança controla todos os acessos aos objetos locais verificando se cada pedido possui uma *capibility* que autorize o acesso. Esta *capability* pode ter sido entregue pelo servidor de autorização, se o acesso for um acesso a um objeto persistente, ou pelo kernel de segurança, se o acesso for um acesso a um objeto local não-persistente. O kernel de segurança pode também reforçar as regras do controle de acesso obrigatório.

Cada kernel de segurança deve reforçar as propriedades usuais de um monitor de referência. Isto é, cada kernel de segurança deve ser invocado e pequeno o bastante para ser analisado para que a integralidade possa assegurada. Deste modo, é confiado a cada kernel de segurança alguma extensão: é muito difícil controlar todos os kernel de segurança, mas se um intruso obter controle de um kernel particular de segurança, não comprometerá a segurança do sistema como um todo, assim o intruso pode controlar todos os acessos aos objetos locais mas não pode receber acesso aos objetos remotos. Cada kernel de segurança do sistema é independente não confiando nos outros kernel de segurança, assim, confiando somente no servidor de autorização.

4.2.3. Controle dos direitos de acesso

Os dois níveis de proteção distinguem a proteção de objetos persistentes e a proteção de objetos locais não-persistentes.

O controle dos direitos de acesso para objetos não persistentes é muito simples, sendo responsabilidade de cada kernel de segurança criar *capabilities* para cada método de um objeto local não-persistente, entregando essas *capabilities* aos objetos que estão autorizados a invocar estes métodos e realizar a verificação quando os métodos são invocados.

A respeito do controle dos direitos de acesso no nível de autorização do usuário, o objetivo é definir uma abordagem ao controle dos direitos de acesso que permite que os privilégios sejam gerados e transmitidos para operações do nível mais elevado do sistema, de forma semi-automática. Os pontos chaves desta abordagem são a simplicidade e a flexibilidade.

4.2.4. Aplicação do modelo multi-nível

O modelo multi-nível faz referência ao modelo Bell-LaPadula explicado no capítulo 2. O principal inconveniente deste modelo é que o mesmo possui propriedades restritivas, não considerando um usuário autenticado como secreto. Deste modo, este usuário pode ler um arquivo confidencial fazendo uma cópia deste arquivo. A propriedade citada requer que o usuário crie uma cópia com uma classificação menos secreta, mas obviamente a classificação da informação que é contida na cópia é a mesma que a classificação da informação do arquivo original. Assim, a propriedade requer que o usuário crie um arquivo com um rótulo que não corresponda à classificação real de seu conteúdo. Este exemplo põe ênfase em um problema, que a classificação da informação no sistema é aumentada vagarosamente, isto conduz ao aumento da degradação da informação no que se diz respeito à acessibilidade.

Um dos pontos fundamentais deste modelo é a distinção de dois grupos de objetos, divididos em objetos sem-estado e objetos persistentes. Todos os objetos de dados são armazenados, mas estes dados podem ser classificados em duas categorias:

- **Dados da aplicação:** são os dados que compõem o estado do objeto, isto é, os dados da aplicação que são declarados, manipulados no código fonte do objeto pelo programador;
- **Dados do sistema:** são dados adicionados pelo sistema no objeto para controlar a execução do objeto (por exemplo, a pilha de processos).

Um objeto sem estado é um objeto que não armazena dados da aplicação: seu estado é inicializado em cada invocação, não havendo nenhum fluxo de informação entre dois pedidos sucessivos que acessem um objeto sem estado. Esta noção do objeto sem estado é análoga à noção do "reusar objeto" do *orange* book. Ao contrário, um objeto persistente é um objeto que armazenem dados da aplicação e cujos métodos consistam na leitura ou na escrita destes dados.

O objetivo da política multi-nível de segurança é prevenir que usuários acessem informações as quais não possuem direito de acesso. Supõe-se que cada tarefa realizada no sistema por um usuário seja realizada por meio de uma atividade. Desta maneira, pode-se impedir fluxos de informação ilegais entre usuários, impedindo os fluxos de informação ilegais entre as atividades. Assim, para assegurar o controle de acesso, tem-se que controlar no sistema todas as interações entre atividades e objetos, realizando o controle de cada pedido que acesse um objeto. A fim de realizar estes controles de acesso, tem-se que atribuir níveis da segurança às entidades diferentes do modelo. Devido aos objetos serem *containers* de informações do sistema, deve-se atribuir um nível de segurança correspondente a sua classificação (rotulando objetos persistentes e objetos sem estado). A fim de controlar cada interação entre atividades e objetos, deve-se atribuir níveis de segurança aos diferentes pedidos do sistema.

Os diferentes níveis de segurança são atribuídos a um objeto através de um rótulo de classificação do objeto. Representando assim a classificação dos dados que compõem o estado do objeto. Esta classificação é fixa e não pode ser mudada durante o tempo de vida do objeto.

Dois rótulos *Low* e *High* são associados a cada objeto sem estado (*Llow0 < Lhigh0). [Llow0, Lhigh0] intervalo* confidencial definindo a confiança que se pode ter sobre cada objeto. Como visto, um objeto sem estado não armazena nenhum dado da aplicação, assim não podendo ser atribuído uma classificação a este objeto. Cada objeto sem estado, pode ser

acessado por uma atividade como um objeto persistente. Cada atividade carrega a informação com um nível particular de segurança, deste modo, cada objeto sem estado pode acessar esta informação. Os rótulos que são atribuídos a um objeto sem estado representam a confiança deste objeto, isto é, as confianças que são colocadas sobre as ações que o objeto. O rótulo *High* representa o nível mais elevado de segurança dos dados que podem ser lidos pelo objeto. O rótulo *Low* representa o nível mais baixo de segurança que os dados podem ser escritos pelo objeto. Por exemplo, um administrador de um sistema que decida utilizar um novo servidor NFS que seja acessado por um usuário anônimo através de um servidor FTP. Então, o administrador estima a confiança que pode colocar neste servidor NFS (que pode ser um objeto sem estado) e atribui-lhe assim um intervalo confidencial que representa esta autorização [NIC97].

4.3. Máquinas virtuais

O conceito de máquina virtual tem várias vantagens. Observe que, neste ambiente, existe proteção dos vários recursos do sistema. Cada máquina virtual é isolada de todas as outras máquinas virtuais, por isso problemas de segurança são menos freqüentes. Por outro lado, não existe compartilhamento direto dos recursos [SIL02].

A máquina virtual tem dois modos de funcionamento: modo usuário e modo monitor (modo privilegiado, indispensável para a realização de atividades críticas do sistema). O software da máquina real executa em modo monitor, pois é o sistema operacional que tem o acesso físico aos recursos da máquina. A máquina virtual propriamente dita, executa no modo usuário. A arquitetura dos sistemas com máquinas virtuais permitem que o sistema em modo monitor (anfitrião) monitore o modo usuário (convidado) realizando o controle das ações através de alguma política de segurança.

Tal sistema de máquina virtual é um método utilizado, por exemplo, em pesquisa e desenvolvimento do próprio sistema operacional real. Normalmente, alterar um sistema operacional é uma tarefa difícil. Como os sistemas operacionais são programas extensos e complexos, é difícil ter certeza de que uma alteração em uma parte não causará erros em outra parte. Essa situação pode ser particularmente perigosa devido ao poder do sistema operacional.

Como o sistema operacional executa em modo monitor, uma alteração errada em uma parte do sistema pode causar danos em outra parte do mesmo. Assim, é necessário testar todas as alterações do sistema operacional cuidadosamente, mesmo assim são constantes os relatos de vulnerabilidades em sistemas operacionais.

Com uso da técnica de virtualização do sistema operacional, mesmo considerando o comprometimento deste sistema virtualizado (onde as aplicações são executadas), o controle da máquina física continuará com o sistema operacional real, o que dá mais segurança à arquitetura como um todo [GAR03. LAU04].

O monitor possui diversas propriedades que podem ser utilizadas na segurança de sistemas:

- **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, que não serão abordadas neste trabalho.

4.4. SELinux

Os resultados de diversos projetos de pesquisa anteriores nesta área foram incorporados no sistema de segurança Linux (SELINUX - Segurança melhorada para Linux). Esta versão do Linux tem uma arquitetura de controle de acesso obrigatório e flexível incorporada nos subsistemas principais do kernel. O sistema fornece um mecanismo para reforçar a separação da informação baseada em exigências de confidencialidade e integridade.

Isto não permite que ameaças (entidades maliciosas) alterem e/ou contornem os mecanismos de segurança da aplicação [LOS98].

Os mecanismos de segurança executados no sistema fornecem suporte flexível para uma ampla gama de políticas de segurança, tornando possível configurar o sistema para reunir diversas exigências de segurança. A flexibilidade do sistema permite que a política seja modificada e estendida para customizar às políticas de segurança necessárias para cada instalação.

O componente de controle de acesso baseado em papel define que cada processo tem um papel associado. Isto assegura que os processos do sistema e aqueles usados para a administração do sistema possam ser separados e atribuídos a usuários comuns. Os arquivos de configuração especificam a lista dos domínios que podem ser incorporados por cada papel.

A arquitetura do SElinux, está baseada na arquitetura *Flask*. Esta arquitetura foi criada na tentativa de servir como uma arquitetura genérica de controle de acesso obrigatório (MAC). Um importante objetivo desse projeto foi prover um suporte flexível para políticas de segurança, uma vez que nenhum modelo de MAC é capaz de satisfazer a uma ampla variedade de requisitos de segurança. Este objetivo foi alcançado pela separação clara entre a lógica da política de segurança e o mecanismo de imposição da política. Cuidados foram tomados para garantir que uma interface de política bem definida fosse especificada e que pudesse suportar um grande conjunto de políticas de segurança. Adicionalmente, com a arquitetura Flask a imposição de uma política de segurança pode ser transparente para as aplicações pelo fato de permitir a definição de comportamento padrão de segurança, na ausência de regras específicas da política.

A arquitetura Flask foi implementada com sucesso a partir da adição da arquitetura de segurança proposta ao sistema operacional Flμke (Flexible MicroKernel), apresentando bons resultados com relação aos objetivos iniciais:

- Segurança;
- Flexibilidade de política;
- Baixo impacto no desempenho;
- Escalabilidade.

A implementação do servidor de segurança para o SELinux demandou a seleção de um modelo concreto de política de segurança, o que levou à escolha de uma combinação de RBAC (Role Based Access Control) e TE (Type Enforcement) [LOS98].

4.4.1. Servidor de segurança

O servidor de segurança é totalmente separado do resto do núcleo do SELinux e isolado através de uma interface bem definida. A flexibilidade da arquitetura Flask permite que o servidor de segurança seja modificado ou substituído para alterar os modelos de segurança suportados. O completo encapsulamento de lógica da política de segurança dentro do servidor de segurança torna possível esse requisito da arquitetura sem causar nenhum impacto ao resto do sistema.

As decisões de segurança tomadas pelo servidor de segurança são baseadas em contextos de segurança (*security contexts*) que representam rótulos de segurança associados a todo sujeito e objeto do sistema operacional. Um contexto de segurança é um identificador independente de política que pode ser manipulado por diferentes partes do sistema, mas deve somente ser interpretado pelo servidor de segurança. Esse contexto contém todos os atributos de segurança associados a um objeto rotulado em particular, e que são relevantes à lógica de decisões da política. No caso específico deste modelo, são três os atributos relevantes à segurança armazenados no contexto de segurança: uma identidade, um papel e um tipo.

4.5. LOMAC (Low water mark)

O LOMAC é um módulo dinâmico de segurança para kernel do Unix ou linux, que tem por finalidade proteger a integridade dos processos e dos dados contra: os vírus, cavalos de tróia e usuários maliciosos. O LOMAC fornece a proteção de um sistema dividindo-o em dois níveis de integridade (alto e baixo). O alto nível de integridade está associado aos componentes críticos do sistema que devem ser protegidos, tais como: processos de inicialização, *daemons* do kernel, bibliotecas e arquivos de configuração [FRA00]. O nível de integridade baixo é associado aos outros componentes, tais como: os processos do cliente e do servidor que oferecem serviços de rede, os processos locais do usuário e suas configurações, como pode ser visto na Figura 4.1.

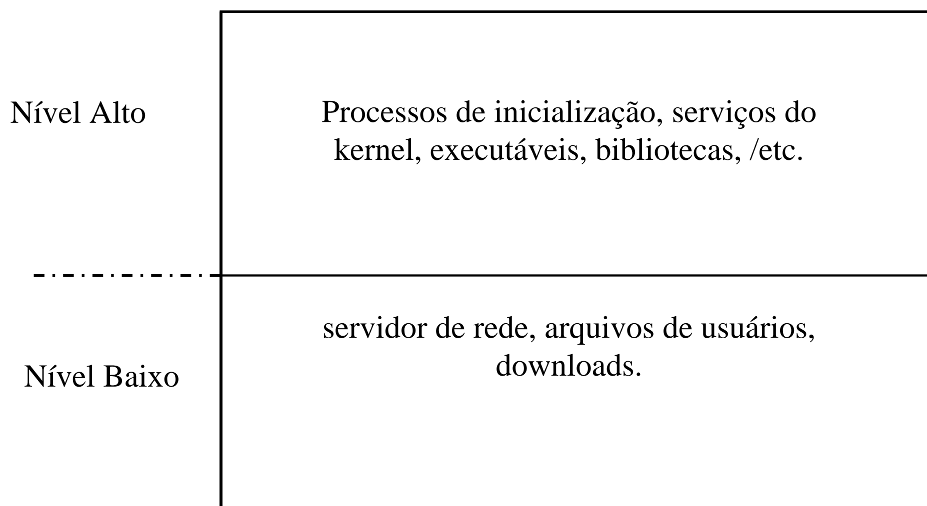

Figura 4.1. – Modelo LOMAC com 2 níveis de integridade

O LOMAC não pode distinguir entre um programa que leia dados de baixa integridade e que ainda está funcionando corretamente de um que leu dados de baixa integridade e foi corrompido. Entretanto, o mesmo assegura que os processos lendo dados de baixa integridade durante a execução podem ser corrompidos.

Há dois pontos principais na execução do MAC (*Mandatory Access Control*) residente no kernel:

- Ganho do controle de supervisão sobre as operações do kernel;
- Mapeamento dos atributos de segurança dos arquivos.

O controle é realizado através da comunicação que o LOMAC estabelece entre os processos. Para fornecer proteção, o LOMAC necessita ter o controle de supervisão sobre operações do kernel.

A interface entre o kernel e o LOMAC é realizada através de uma série de funções chamadas *wrappers* (interceptadores). Haverá um interceptador para cada chamada (relevante) do sistema à segurança do Linux. Além de ganhar o controle de supervisão, o LOMAC também deve atribuir níveis de integridade aos arquivos de modo que seja persistente através das reinicializações do sistema [FRA01].

O LOMAC deve determinar o nível apropriado para cada processo e a configuração do sistema e também possui habilidade de atribuir automaticamente os níveis apropriados aos usuários e aos servidores da rede sem configuração local específica.

O LOMAC impede que os processos de baixa integridade modifiquem os arquivos de alta integridade. Somente quando um processo de baixa integridade adquire privilégios de *root*, há uma diferença aparente: um processo de baixa integridade do *root* reduz extremamente seus poderes na presença de LOMAC.

Na inicialização o LOMAC atribui um nível alto de integridade ao primeiro processo, criando um novo processo de nível alto de integridade para várias tarefas do sistema. Este comportamento permite que o LOMAC atribua automaticamente as sessões de usuário, o nível de integridade apropriado para cada usuário. O LOMAC trata todas as relações da rede como arquivos de nível baixo de integridade. O esquema de proteção do LOMAC é projetado especificamente para impedir que processos transfiram dados de baixa integridade para alta integridade.

O mecanismo de proteção do LOMAC não depende do mecanismo baseado na identidade do superusuário existente na proteção do kernel do Linux. O LOMAC fornece a proteção observando pedidos para o serviço feito por processos na relação das chamadas de sistemas do kernel, negando os pedidos que são identificados como ameaças à integridade do sistema.

O LOMAC ainda não está otimizado, havendo diversas áreas de sua execução nas quais o desempenho não é o seu melhor ponto, mas tem suporte ao desenvolvimento rápido de novas características. O LOMAC ainda não controla todas as operações críticas do kernel e não fornece nenhuma proteção para o nível baixo de integridade do sistema [FRA01].

4.5.1. Proteção de integridade

O protótipo do LOMAC implementa uma forma de controle de acesso baseado em uma versão estendida do modelo da marca d´agua baixa do sujeito. O LOMAC atribui um nível a cada objeto existente, dividindo todos os objetos nas classes baseadas em seu nível de integridade. Uma vez atribuídos, os níveis dos objetos nunca mudam. O LOMAC fornece

uma proteção de integridade para prevenir o movimento e a corrupção dos dados de nível baixo ao nível alto dos objetos restringindo o comportamento dos sujeitos.

Na configuração do LOMAC, cada instalação pode definir sua própria política. Uma política especifica o número de níveis a serem utilizados, é a forma de mapear os objetos entre os níveis existentes. Em outros casos o LOMAC pode implementar uma política padrão que fornece um nível padrão de proteção de integridade apropriado para todos os ambientes. Esta política padrão contém somente dois níveis:

1. Para objetos de baixa integridade, tais como os conteúdos de download da Internet;
2. Nível utilizado para objetos com alto nível de integridade, tais como os arquivos com código executável do sistema.

4.6. Separação de privilégios de execução

A separação de privilégios aplica-se a todo o serviço executando no modo supervisor nos sistemas operacionais baseados em Unix. Tais serviços são difíceis de confinar porque seu estado interno não é conhecido pelo sistema de confinamento da aplicação, por essa razão não é possível restringir as operações que o serviço pôde executar para usuários autenticados. Como conseqüência, um intruso (sujeito não autorizado) que ganha o controle de um serviço pode executar as mesmas operações que os usuários autorizados. Com separação do privilégio, o adversário controla somente a parte sem privilégios do código e não obtém nenhum privilégio desautorizado. A separação de privilégios facilita também os exames de código fonte, reduzindo a quantidade de código que necessita ser inspecionado intensivamente [PRO03].

4.6.1. Menor privilégio

Um privilégio é referenciado como um atributo requisitado para executar determinadas operações. O princípio de menor privilégio define que cada programa e cada usuário, deve trabalhar usando o menor conjunto de privilégio necessário pra executar uma operação.

Nesta aplicação são sugeridas três abordagens ao projeto da aplicação que ajudem, impedindo conseqüências não antecipadas de tais erros: programação defensiva, proteção reforçada de linguagem e mecanismos de proteção suportados pelo sistema operacional. As duas últimas abordagens não são aplicáveis a muitos sistemas baseado em Unix porque os mesmos são desenvolvidos em linguagem C.

Os privilégios que são relacionados a acesso do sistema de configuração granularidade fina porque o sistema concede o acesso baseado na identidade do usuário e permissões do grupo. Em operações gerais, privilégios são executados através das chamadas do sistema no kernel do Unix, que se diferencia principalmente entre o super usuário e os demais.

A probabilidade dos erros é elevada, e um intruso poderá usar erros internos para ganhar o privilégio especial. Mesmo se o princípio de menor privilégio for seguido, um intruso pode ganhar os privilégios que são necessários para a aplicação funcionar.

4.6.2. Separação de privilégio

A separação de privilégio é ortogonal a outros mecanismos de proteção que um sistema operacional possa suportar, por exemplo, capabilities ou domínios de proteção. É descrita uma execução da separação do privilégio que não requer suporte especial do kernel do sistema operacional, podendo ser executada em quase todos os sistemas operacionais baseados em Unix. O objetivo da separação do privilégio é reduzir a quantidade de código que funciona com privilégio especial, desta forma algumas partes funcionam com privilégios e as outras sem os mesmos. A parte privilegiada é o monitor e as peças sem privilégios são os escravos.

Quando existir somente um escravo (não é uma exigência), este deve pedir que o monitor execute toda a operação que requer privilégios. Antes de atender a um pedido do escravo, o monitor precisa validá-lo. Se o pedido for permitido, o monitor executa-o e comunica os resultados ao escravo.

A fim de separar os privilégios em um serviço, é necessário identificar as operações que as requerem. O número de tais operações é geralmente pequeno comparado às operações que podem ser executadas sem privilégio especial. A separação do privilégio reduz o número

dos erros de programação que ocorrem em um trecho privilegiado do código. Além disso, os exames do código fonte podem focar no código que é executado com privilégio especial, o que pode reduzir a incidência de escalonamento desautorizado de privilégio.

Embora os erros na parte sem privilegio do código não pudessem resultar em nenhum escalonamento de privilégio, podem ainda ocorrer outros ataques com interrupção de recursos.

Os processos são domínios de proteção em um sistema Unix. Isso significa que um processo não pode controlar um outro processo sem relacionamento. Para conseguir a separação do privilégio, foram criadas duas entidades: um processo pai privilegiado que age como o monitor e um processo filho sem privilégios que age como o escravo. O processo pai aceita pedidos do processo filho para as ações que necessitam de privilégios. Se o número de ações que necessitam de privilégios for pequeno, a maior parte do código da aplicação está sendo executado pelo processo filho sem privilégios. O projeto de relação é importante porque fornece prevenção do ataque de um intruso que queira comprometer o processo filho.

O serviço de separação de privilegio pode estar separado em duas fases:

- Pré-Autenticação: Um usuário contatou um serviço do sistema, mas ainda não se autenticou. Neste caso, o processo filho não tem nenhum privilégio do processo e nenhum direito de acessar o sistema de arquivos;
- Pós-Autenticação: O usuário se autenticou com sucesso no sistema. O filho tem os privilégios do usuário incluindo o acesso ao sistema de arquivos, mas não possuiu nenhum outro privilégio especial. Entretanto, o privilégio especial é requerido para executar outras operações privilegiadas. Para estas operações, o processo filho deve pedir uma ação do processo pai privilegiado.

4.6.3. Separação de privilégios SSH

Um exemplo de utilização da separação de privilégio é demonstrada utilizando o *OpenSSH*, uma versão livre do protocolo *SSH*. O *OpenSSH* fornece uma autenticação remota segura através da Internet. O *OpenSSH* suporta versões de protocolo um e dois; este exemplo

é restrito a separação de privilégio da versão dois. O procedimento é muito similar para o protocolo um e se aplica também a outros serviços de autenticação.

Quando o *daemon* do SSH é solicitado, o mesmo inicia um soquete para abrir a porta 22 e esperar novas conexões. Cada nova conexão é mantida por um processo filho. O processo filho necessita possuir privilégios de superusuário durante todo o seu ciclo, criando novos pseudoterminais para o usuário, autenticando trocas de chave quando as chaves cifradas são substituídas, removendo pseudoterminais quando a sessão de SSH termina, criando processos com privilégios de autenticação de usuário, etc.

Com a separação do privilégio, o processo filho age como monitor e gera um processo escravo que fica sem os seus privilégios começando a aceitar os dados da conexão estabelecida.

Em qualquer tempo, o número de pedidos que o SSH escravo pode enviar é limitado. Quando o SSH monitor é iniciado, o SSH escravo primeiramente pode enviar somente dois pedidos. Depois que a troca de chave foi concluída, o único pedido aceito é para a validação do usuário. Após ter validado o usuário, todos os pedidos de autenticação são permitidos. A motivação para manter o número reduzido de pedidos é para prevenir um ataque por intrusos que queiram comprometer o processo do SSH escravo [PRO03].

A fase de pré-autenticação termina com a autenticação bem sucedida determinada pelo SSH monitor. Neste momento, o processo SSH escravo necessita mudar sua identidade para a identidade do usuário autenticado. Em conseqüência, o escravo obtém todos os privilégios do usuário, mas nenhum outro privilégio.

4.7. Outros mecanismos utilizados para prover segurança

As seções anteriores deste capítulo apresentaram alguns modelos de segurança que se baseiam no controle de acesso obrigatório, citando os benefícios da implantação deste modelo em sistemas operacionais. Abaixo será vista uma breve introdução sobre alguns mecanismos que tem funcionalidade semelhante ao LOMAC.

4.7.1. Janus

O Janus é uma ferramenta para confinar processos potencialmente perigosos, tais como usuários privilegiados e aplicações WEB. O Janus foi desenvolvido originalmente no Solaris, utilizando o espaço do usuário dos sistemas operacionais para correções de processos. Existe versão para Linux [WAG99].

4.7.2. Kernel Hypervisors

O kernel *hypervisors* possui módulos carregáveis "*wrap*", que tem a finalidade de substituir um subconjunto de interfaces de chamadas do sistema do kernel com troca destas interfaces pelas suas próprias. Esta troca pode filtrar as chamadas a fim executar o controle de acesso, ou pode incrementar as chamadas de sistemas com nova funcionalidade. Além disso, o kernel *hypervisors* [MIT97] é uma ferramenta geral que pode ser usada para executar diversas funcionalidades de interface de segurança.

4.7.3. Generic Software Wrappers Toolkit

Este projeto genérico do laboratório NAI produziu um conjunto de ferramentas para construir uma segurança abstrata melhorada chamada *wrappers* [FRA99], que possui a finalidade similar ao kernel *hypervisors.* O conjunto de ferramentas fornece uma abstração do ciclo de vida controlando os relacionamentos entre *wrappers* e processos, e uma linguagem para escrita de *wrappers*. O conjunto de ferramentas suporta atualmente Linux, FreeBSD, Solaris, e (a alguma extensão) Windows NT.

4.7.4. Medusa DS9

O Medusa DS9 [ZEL97] fornece um mecanismo para implementação geral de funcionalidades de segurança, incluindo o MAC. É executado como um pequeno *patch* do kernel do Linux, fornecendo uma relação do serviço a um *daemon* de segurança que funciona no espaço do usuário. Este *daemon* de segurança executa a maioria das funcionalidades desejadas de segurança, mantendo estes detalhes fora do kernel. Atualmente, um *daemon* de segurança chamado "*constable*" foi implementado e age como um servidor de autorização.

4.7.5. Remus

O Remus é um módulo carregável que faz a interposição na relação das chamadas do sistema para executar sua funcionalidade de controle de acesso, e pode ser aplicado ao kernel do Linux [BER02]. O Remus foi projetado para confinar usuários privilegiados e processos com *setuid* que fazem uso de chamadas do sistema relevantes ao sistema de segurança. Os administradores podem configurar o Remus para permitir que os processos confinados façam somente determinadas chamadas com determinados parâmetros.

4.7.6. VXE

O VXE reforça o princípio de menor privilégio confinando processos aos ambientes virtuais em que somente o conjunto mínimo de recursos para a operação normal está disponível. Os ambientes virtuais de VXE são executados em módulos carregáveis e funcionam com o reforço de um *patch* do kernel do Linux. Desde que estes ambientes virtuais podem ser especificados às necessidades dos recursos de uma aplicação particular, o VXE fornece suporte para controle de granularidade fina a aplicações [ODE04].

4.8. Conclusão

Os mecanismos de controle de acesso obrigatórios estão sendo pesquisados e desenvolvidos cada vez mais para prover segurança em sistemas computacionais. A base destes mecanismos propostos parte na maioria das vezes, dos modelos de confidencialidade Bell-Lapadula (BLP) e do modelo de integridade BIBA [AMO94], os quais foram abordados no capítulo 2 desta dissertação. Os principais mecanismos descritos neste capítulo foram SElinux que visa a confidencialidade baseado no modelo BLP e o LOMAC que visa a integridade baseado no modelo BIBA. Outros assuntos relevantes deste capítulo foram as abordagens de artigos sobre separação de privilégios, modelo de autorização multí-nível (objeto sem estado), segurança de kernel, modelo do cofre entre outras abordagens no sentido de mecanismos de segurança.

Capítulo 5

Provendo Comportamento Discricionário ao Modelo BIBA de Integridade

Com o surgimento de novas tecnologias, tanto de segurança quanto para ataques a sistemas computacionais, se torna cada vez mais difícil a tarefa de se proteger um sistema bem como garantir a integridade de seus dados. Neste capítulo é proposto um mecanismo que irá fazer com que a transição entre diferentes níveis de integridade imposta pelo LOMAC seja possível sem a necessidade da utilização de sujeitos de confiança. O LOMAC cria exceções para sujeitos de confiança, o que pode comprometer a segurança do sistema como um todo. O LOMAC não prevê alteração de conteúdos de alta integridade por usuários de baixa integridade mesmo que o conteúdo pertença a este usuário. O modelo proposto visa garantir a integridade dos dados de um sistema operacional sem a utilização de sujeitos de confiança; não violando as regras impostas pelo modelo BIBA.

5.1. Problema

O principal problema encontrado no LOMAC foi que o mesmo foi criado para garantir a segurança de um sistema operacional no que se refere à integridade, mas devido a algumas limitações de sua implementação, o mesmo pode se tornar suscetível a ataques.

O LOMAC tem como seu objetivo a qualidade da proteção do controle de acesso obrigatório e a compatibilidade. A primeira função do LOMAC é ser compatível com o software existente no sistema operacional e fornecer proteção de integridade baseada no modelo obrigatório BIBA. O modelo da marca d‘água baixa suporta esta primeira exigência

de compatibilidade; entretanto, a qualidade de proteção que é fornecida não é tão boa quanto àquela fornecida pelo modelo clássico, pois apresenta menor compatibilidade [FRA00, FRA01].

Um inconveniente do modelo de marca d'agua baixa do sujeito é o do princípio de menor privilégio, onde o "bom" esquema do controle de acesso obrigatório deve conceder a um sujeito um conjunto mínimo de privilégios necessários para realizar o seu trabalho [SAL75]. Deste modo, se um sujeito for confinado, será minimizado a quantidade de danos que o mesmo pode causar ao sistema, caso esse venha a ser comprometido. O LOMAC trata os usuários de rede, rebaixando o seu nível para um nível baixo de integridade; desta maneira, protege a parte do sistema que está em nível alto de integridade. Embora o LOMAC impeça que um usuário mal intencionado em nível baixo de integridade cause algum dano ao nível alto de integridade, o mesmo não pode impedir que o usuário mal intencionado cause danos no restante do sistema que está no nível baixo de integridade.

Outro inconveniente do LOMAC é a sua dependência em sujeitos de confiança, os quais são necessários para que o sistema possa funcionar normalmente. Esta dependência é uma característica de modelos MAC como o modelo de marca d'agua baixa [FRA00].

O Linux registra eventos do sistema no arquivo */var/log/messages*, através do *daemon* de *syslog* (*syslogd)*, que trabalha lendo um soquete do registro de sistema, */dev/log*, o qual possui informações de eventos do sistema. Os processos com nível baixo de integridade possuem permissão para escrever em */dev/log*, a fim de registrar suas operações no sistema. Conseqüentemente, a política padrão do LOMAC atribui a /dev/log um baixo nível de integridade. Verificando a descrição do comportamento do LOMAC, o mesmo teria que rebaixar o *syslogd* para nivelar em nível baixo de integridade o seu primeiro acesso ao registro do soquete */dev/log*. Se o *syslogd* estiver funcionando em nível baixo de integridade, então o */var/log/messages* também deve estar em nível baixo, a fim de permitir que o *syslogd* armazene suas mensagens no arquivo de registro do sistema.

Entretanto, os usuários maliciosos freqüentemente procuram apagar rastros gravados no arquivo de registro do sistema, para não terem identificada as suas ações no sistema. A fim de impedir que isto aconteça, o LOMAC atribui ao */var/log/messages* um alto nível de integridade. Mas esta atribuição causaria um outro problema. O *syslogd* estando em um nível

baixo de integridade não poderia escrever em um objeto que esteja em um nível alto de integridade (*/var/log/messages*).

O LOMAC tenta solucionar este problema adicionando o *daemon* do registro de sistema (*syslogd*) em sua lista de programas "confiáveis". Um processo que funcione como um sujeito de confiança ganha um privilégio extra; deste modo um processo que esteja em um nível alto de integridade, não irá rebaixar o seu nível ao acessar um objeto em nível baixo de integridade. Esta exceção permite que o *daemon* de registro de eventos do sistema (*syslogd*), funcionando em um nível alto de integridade leia o */dev/log,* que está em nível baixo de integridade sem rebaixar o seu nível de integridade, escreva em */var/log/messages* que está em um nível alto de integridade.

A utilização do conceito "*trust subject*" significa descrever que os processos que foram verificados formalmente forneçam garantia de que não irão abusar de seus privilégios adicionais. Infelizmente, não se pode ter essa garantia para o *syslogd*. Se um usuário malicioso encontrar alguma forma de comprometer o *syslogd*, talvez explorando um *buffer overflow* ou algum outro erro, poderá ganhar o controle do sistema em nível alto de integridade, equivalente a um privilégio de superusuário e desta forma comprometerá todo o sistema.

O LOMAC também não prevê a alteração de conteúdos de alto nível de integridade por usuários de baixo nível de integridade mesmo que o conteúdo pertença a este usuário, gerando uma dependência do administrador do sistema para que um usuário que esteja em um nível baixo de integridade possa alterar as suas informações que estão em um nível alto de integridade. Um exemplo para este caso é a alteração de senhas de sujeitos com nível baixo de integridade, os mesmos não podem alterar a sua senha, pois as mesmas estão guardadas em um local que está em nível alto de integridade; desta forma a operação de escrita em um objeto que esteja em nível alto de integridade por um sujeito que esteja em nível baixo de integridade não é possível devido à regra do modelo BIBA (NWU), tornando-se inviável a alteração da senha, mesmo o conteúdo pertencendo ao próprio usuário.

5.2. Proposta

A proposta deste trabalho é realizar a configuração do LOMAC no Linux de forma confiável, tratando a exceção que o mesmo faz para o *daemon syslogd* e permitir a alteração de conteúdos pertencente ao usuário mesmo que o usuário e o conteúdo estejam em diferentes níveis de integridade. Inicialmente será abordado o caso do arquivamento de registros de eventos no *log* do sistema. O caso de alteração de senhas será tratado na seção 5.6.

O problema em questão é relacionado ao não rebaixamento do nível de integridade do serviço que realiza o registro de informações do sistema (*syslogd*). O referido serviço realiza a leitura das informações contidas em um dispositivo que possui um baixo nível de integridade e grava estas informações em um arquivo de armazenamento que possui um alto nível de integridade. Considerando que o serviço de registro possui um alto nível de integridade seria impossível o mesmo realizar tal tarefa, se não fossem violadas as regras do modelo (NRD). A Figura 5.1 demonstra a exceção efetivada no LOMAC ao *daemon syslgod*, onde o mesmo funcionando em um nível alto de integridade (nível 2) faz a leitura de um objeto (*/dev/log*) que está em nível baixo de integridade (nível 1) e grava as informações lidas em um objeto (*/var/log/messages*) que está em nível alto de integridade(nível 2) , não sendo rebaixado o nível de integridade do *syslgod*.

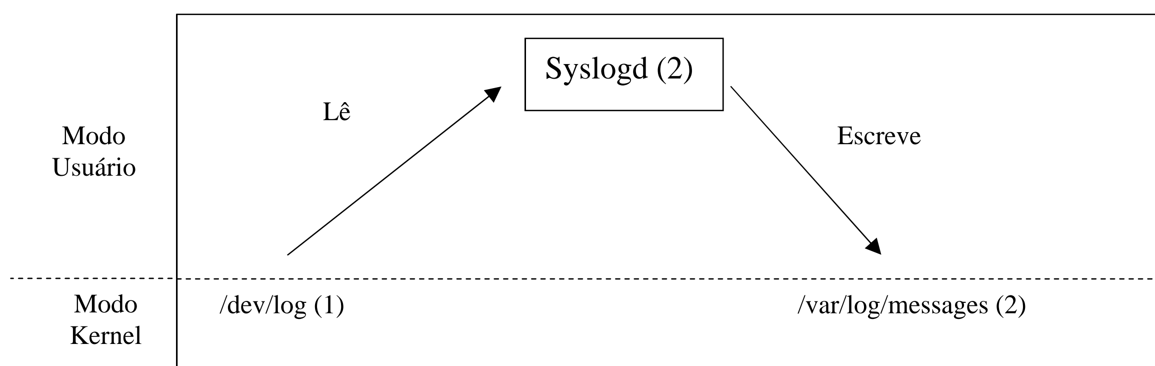

Figura 5.1. – Funcionamento do LOMAC para o syslogd

A reformulação proposta caracteriza-se em manter a propriedade simples de integridade do modelo BIBA, onde não é permitida diretamente a ação descrita na Figura 5.1, que se torna possível devido à exceção realizada ao serviço de registro de eventos (*syslogd*). A proposta será melhor detalhada no item 5.4.

5.3. Recursos utilizados na proposta

Para se obter um melhor entendimento da proposta; serão detalhadas nos itens subseqüentes, as tecnologias adotadas para implementação da mesma, no que se refere ao tratamento do problema de gerenciamento multi-nível do LOMAC para sua implementação no Linux, não sendo totalmente compatível com o modelo de integridade do BIBA.

5.3.1. Repositório de registros

Sugere-se a utilização de um repositório de dados, onde serão armazenadas as informações geradas pelos serviços de registro de eventos do sistema (*KTlog* – gerenciador de logs baseado em kernel *threads*) e também do kernel (*klogd*). Partindo do modelo de implementação do LOMAC utilizado na proposta, o repositório de registros terá um nível de integridade superior ao nível de integridade alto (2), um meta nível de integridade (3 = 2 + 1). O nível 3 foi criado por que o repositório de registros armazena os registros de informações de baixo e alto nível de integridade. Os dois tipos de registros (1 e 2) como mostrado na Figura 5.2 serão armazenados em arquivos separados; messages1 e messages2, respectivamente. Tal opção de projeto foi adotada em função de se ter a dificuldade de implementação da separação dos tipos de registros (1 e 2) em um mesmo arquivo do tipo texto.

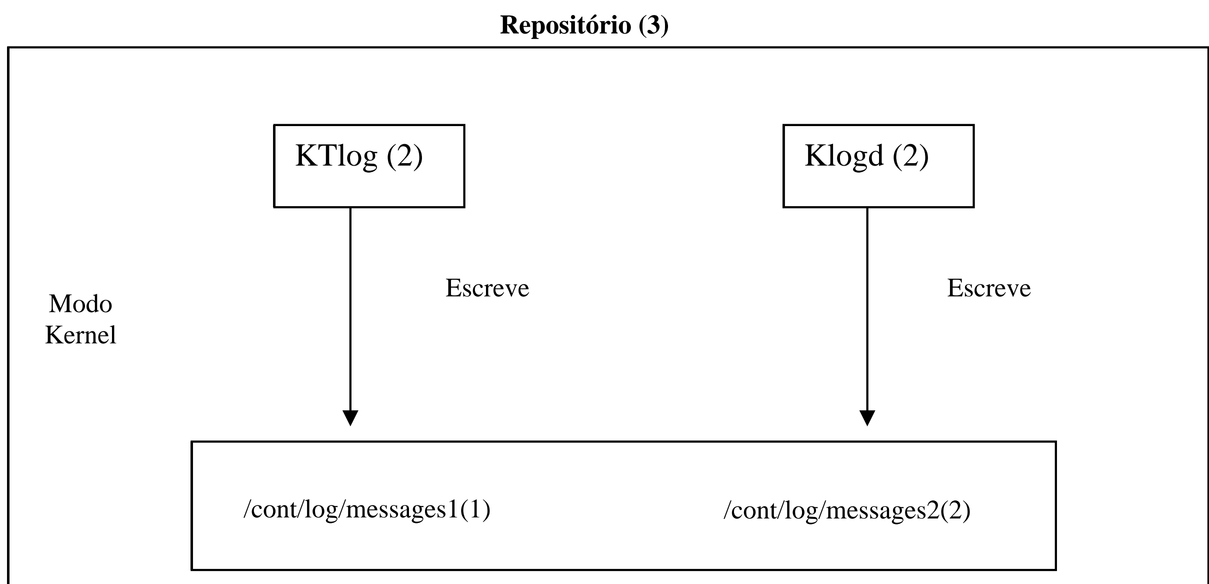

Figura 5.2. – Repositório de nível de integridade 3 armazenando dados de nível de integridade 1 e 2.

5.3.2. Dispositivo de transição

A criação de um dispositivo de transição foi fundamental para a solução do problema de transição entre diferentes níveis de integridade. A utilização de um objeto sem estado (OSE) para a criação do dispositivo de transição se fez necessária, pois um objeto sem estado não armazena dados de aplicação (persistente). O estado de um objeto OSE é inicializado a cada invocação, não havendo nenhum fluxo de informação entre dois pedidos sucessivos que acessem um mesmo OSE. Um OSE não armazena nenhum dado de aplicação, logo não se pode atribuir uma classificação a este objeto, pois não é persistente. Como não se pode atribuir nível de integridade a um OSE porque não é possível ganhar o controle do mesmo, o OSE se mostrou um excelente mecanismo para permitir que um sujeito de baixo nível de integridade possa guardar dados em um repositório de alto nível de integridade. Por exemplo, como no mundo físico faz um indivíduo que deseja guardar algo de valor em um cofre. O que não é possível de ser realizado pelas regras do BIBA. Para arquivos executáveis as regras do modelo de integridade BIBA tem sentido, mas para o caso de armazenamento de informações em formato texto, como é o caso dos registros de *log*, o comportamento do LOMAC não tem sentido. Ou seja, não se pode proibir um sujeito de baixo nível de integridade de escrever informações em formato texto de maneira seqüencial.

Para realização da proposta do dispositivo de transição, foram estudados os conceitos sobre soquetes e *pipes,* os quais demonstraram-se inviáveis para a proposta, devido ao problema de interceptação de comunicação, já que os mesmos criam um canal de comunicação para o fluxo da informação entre transmissor e receptor, que pode ser controlado por um principal ou eventual intruso. A estratégia então foi utilizar um *device driver* do tipo *character*.

5.3.3. Serviço de *Log*

O serviço de *log* criado possui a finalidade de ler de um arquivo (objeto sem estado) e gravar em outro arquivo com alto nível de integridade. Este serviço irá funcionar em modo kernel, por este motivo o mesmo foi criado utilizando o conceito de kernel *thread.* Funcionando como um serviço de kernel, o mesmo terá um alto nível de integridade imposto pelo LOMAC e não poderá ser acessado por processos que estejam nível baixo de integridade. Outro detalhe importante na proposta deste serviço de *log* é que o mesmo utilizará chamadas

de sistema para realizar a leitura e escrita dos arquivos descritos. As chamadas de sistema utilizadas são: *open, sendfile e close*.

Como visto no capítulo 4, que detalha o funcionamento do LOMAC, o mesmo intercepta as chamadas de sistema para obter um melhor controle de integridade, e a proposta sugerida utiliza a chamada de sistema *sendifle* que não é interceptada pelo LOMAC, mas que poderia ser interceptada, pois a criação do módulo deste serviço é realizada antes do modulo do LOMAC.

A utilização da chamada de sistema *sendfile* torna-se uma melhor alternativa no caso da utilização das chamadas de sistemas *read* e *write*, pois para utilizar estas duas chamadas de sistema seria necessário criar um *buffer* de transição interno, o que poderia causar problemas como (*buffer overflow*), e conseqüentemente o comprometimento do sistema como um todo. Já que o serviço de *log* está funcionando em modo kernel.

5.4. Dinâmica do mecanismo proposto

Já vimos no capítulo anterior que o LOMAC tem seu funcionamento em dois níveis de integridade (baixo e alto), e este funcionamento é baseado no modelo BIBA de integridade. Desta maneira, fica claro que:

- Um processo de alto nível de integridade terá seu nível de integridade rebaixado caso o mesmo acesse um objeto de baixo nível de integridade, o que não acontece com o *daemon syslgod*, devido à exceção criada na implementação do LOMAC;
- Um processo de baixo nível de integridade só pode ler um objeto de alto nível de integridade, não podendo alterá-lo de maneira alguma, garantindo assim a sua integridade.

Como visto nos itens acima, a exceção criada pelo LOMAC, gera um certo nível de incompatibilidade no seu modelo de implementação, pois caso um processo de nível (1) consiga ganhar o controle do *syslogd* (2) que não rebaixa seu nível acessando o /dev/log (1), toda a integridade do sistema estará em risco.

Devido ao caso citado (*syslogd*), se propõe a utilização de duas novas peças de software, as quais terão a finalidade de rever a exceção criada pelo LOMAC, tratando a exceção entre níveis de prioridade de forma mais compatível com o modelo BIBA de integridade.

5.4.1. Dispositivo baseado em OSE

O dispositivo baseado em OSE do modelo proposto será criado a partir de um dispositivo de caractere que segue o comportamento de um objeto sem estado, visto na seção anterior. A proposta do dispositivo baseado em OSE terá a finalidade de ser um objeto de transição entre o nível 1 e nível 2. Ao objeto de transição não é associado nenhum nível de integridade. O dispositivo em */dev/ddLOMAC*, implementa o objeto de transição e pode ser lido ou escrito por qualquer processo do sistema, pois não possui nenhum nível de integridade associado.

O objeto de transição terá um controle de acesso que somente o processo existente *syslogd* (1) fará a escrita no dispositivo criado e somente o processo *KTlog* (2) (Gerenciador de logs baseado em kernel *threads*) fará a leitura do mesmo. A Figura 5.3 demonstra a implementação do objeto de transição no esquema proposto para o caso do *syslogd* em um sistema que usa o LOMAC.

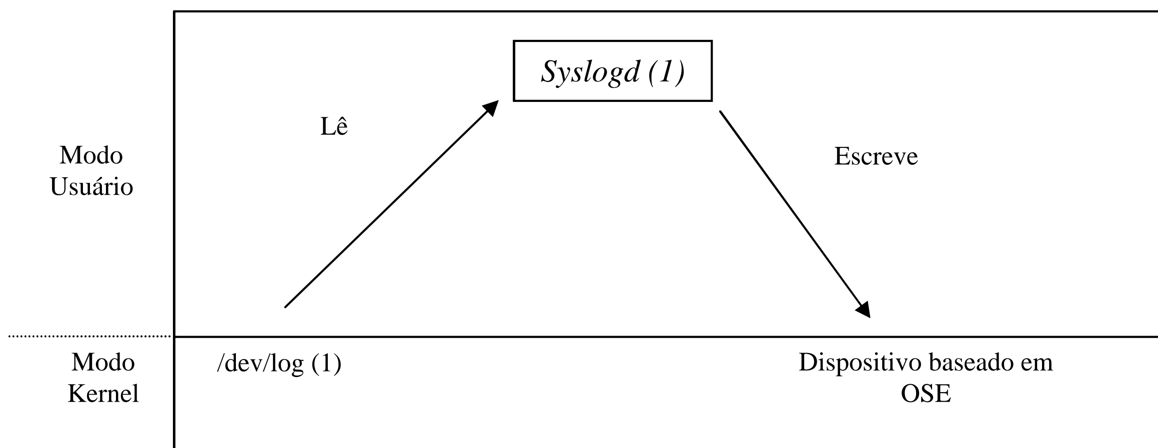

Figura 5.3. – Modelo de utilização do dispositivo baseado em OSE

5.4.2. Serviço *KTlog*

Devido à necessidade de se ter um serviço para realizar a transferência de informações do objeto de transição para um objeto de nível (2), foi desenvolvido o serviço *KTlog*. O serviço proposto está baseado no conceito de threads que trabalham no espaço do kernel, para escrever em um objeto de nível (2) é necessário estar no mesmo nível de integridade. Além disto, executando em nível de kernel esta *thread* apresenta-se melhor proteção de ataques e executando a alteração indevida do arquivo de *log*. O serviço *KTlog* realizará a leitura dos eventos do sistema que estarão gravados no objeto de transição e fará a escrita destes registros no arquivo */cont/log/messages1(2)*, Figura 5.4.

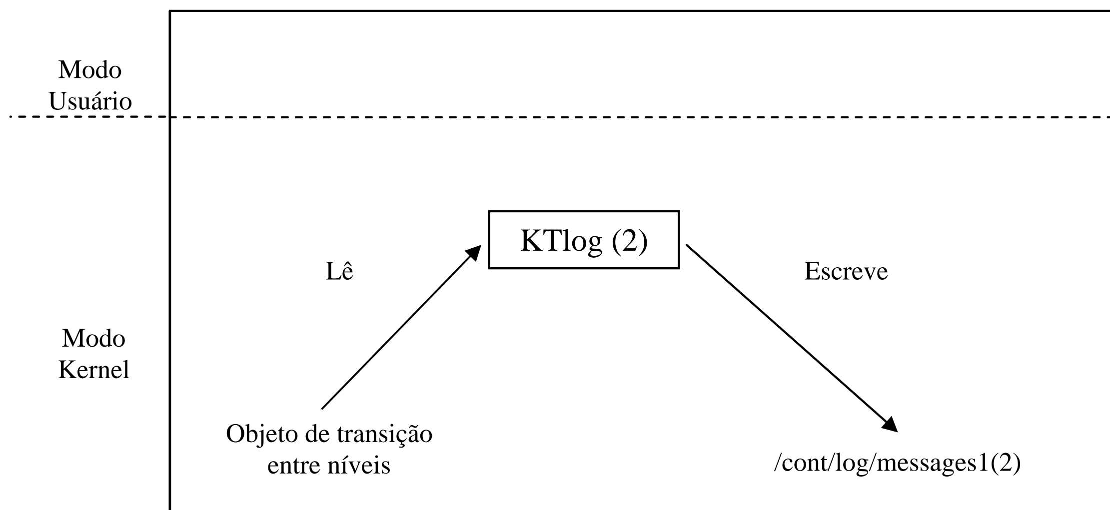

Figura 5.4. – Modelo de utilização do serviço KTlog

5.5. Detalhamento estruturado da proposta

Para realização da proposta em um contexto geral, foram feitas alterações em um arquivo de configuração de *logs (/etc/syslog.conf)* do Linux e também a modificação do código do LOMAC para obter um nível de integridade (3) que permitisse armazenar os registros dos eventos do sistema (nível 1) e também os registros de eventos do kernel (nível 2).

O registro de eventos do sistema, gerados por processos usuário e manipulados pelo *daemon syslogd,* são gravados no objeto de transição */dev/ddlomac* entre níveis e coletados pelo serviço proposto *KTlog* e em seguida gravadas no arquivo *(/cont/log/messages1)* de nível

(2); o arquivo *messages1* conterá somente informações sobre os eventos de processos do usuário. Os regristros de eventos do kernel gerados pelo *daemon klogd* serão gravados diretamente no arquivo *(/cont/log/messages2)*, contendo somente registros de eventos do kernel. Desta maneira o diretório "cont" (repositório de informações) criado e alterado no LOMAC terá o nível (3) de integridade. O diretório "log" terá o nível (2) de integridade e guardará informações dos eventos do usuário e do kernel, Figura 5.5.

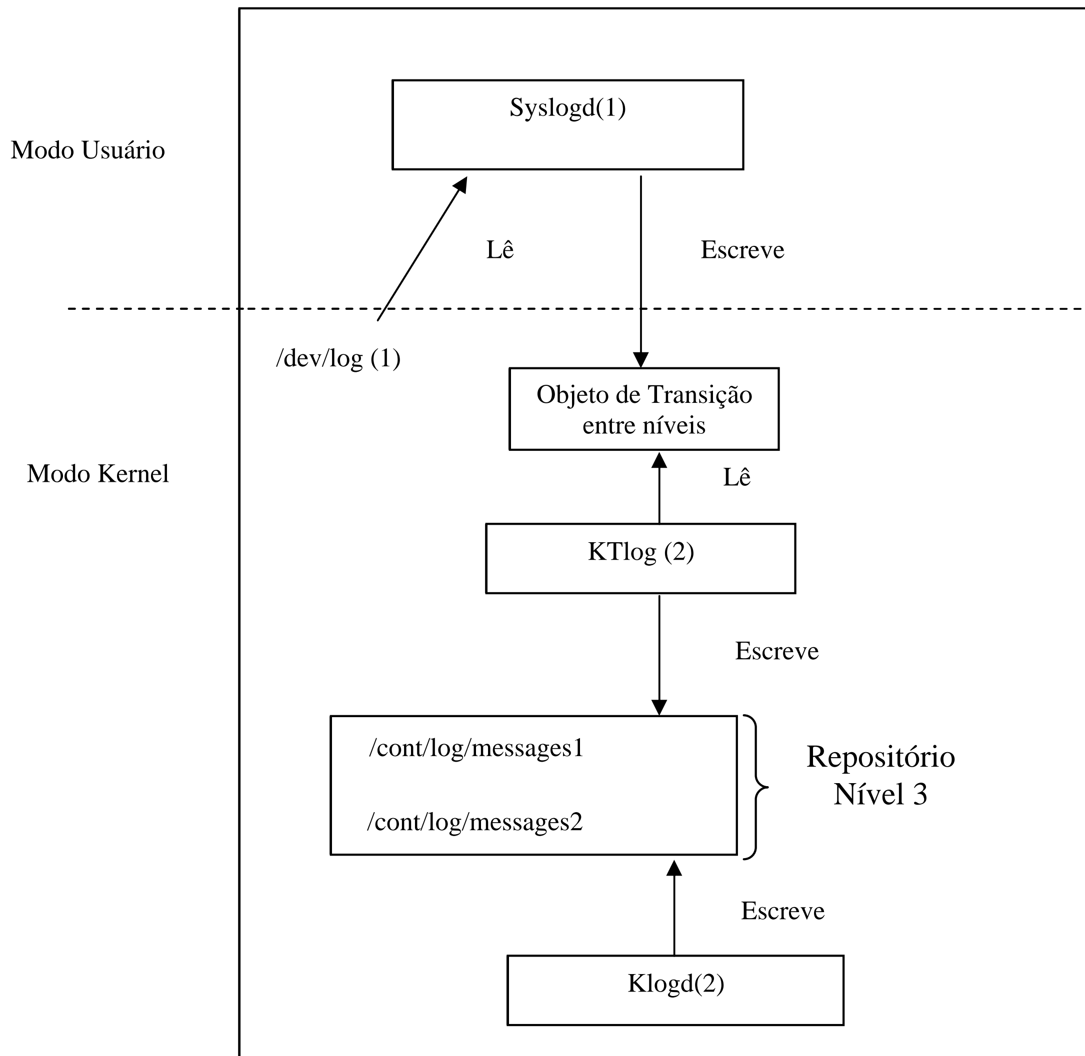

Figura 5.5. – Modelo Geral da Proposta

Para evitar a paralisação maliciosa do serviço de registro de eventos do sistema, adotou-se a estratégia de priorizar o registro de eventos do kernel. Ou seja, se o objeto de transição ou o serviço *syslogd* forem atacados por negação de serviço, o serviço de registro de eventos do kernel (*klogd*) será priorizado e continuará funcionando. Como o *klogd* executa em

modo kernel e a *thread* de registro de eventos *KTlog* também, o sistema de *log* deverá continuar operando, mesmo nesta situação.

5.6. Problema da troca de senhas

Outra incompatibilidade do LOMAC é a não possibilidade do usuário sem permissão de superusuário trocar sua própria senha. Isto acontece devido a chamada do serviço para troca de senha ser de baixo nível de integridade (*passwd(1)*) e o usuário estar nível baixo de integridade, porém o arquivo de armazenamento de senhas é de alto nível de integridade (2). Desta maneira, um sujeito de nível 1 não pode ler um objeto de nível 2 e esta ação se torna inviável na presença do LOMAC.

5.6.1. Proposta para troca de senhas

A proposta de solução para o problema da troca de senhas esta baseado na separação de privilégios citada no capítulo 4.

Para que um sujeito possa alterar informações que lhe pertencem, mas que estão em nível integridade superior ao seu, propõe-se a codificação de um programa com separação de privilégios. Deste modo o sujeito em baixo nível de integridade que realize um pedido para troca de senhas (alto nível de integridade), fará este pedido ao processo pai que realizará internamente a tarefa solicitada e enviará um retorno ao solicitante informando o status da execução, Figura 5.6.

A proposta sugerida é de que um ambiente com controle de acesso obrigatório possa prover um comportamento discricionário a determinados sujeitos do sistema. Este comportamento discricionário se deve ao fato de que um sujeito estando em baixo nível de integridade possa alterar informações que lhe pertencem e estão armazenadas em um repositório de alto nível de integridade.

A referida proposta não foi implementada, devido ao tempo disponível para o termino deste trabalho, já que foi implementada a proposta de solução do problema de transição entre diferentes níveis de integridade.

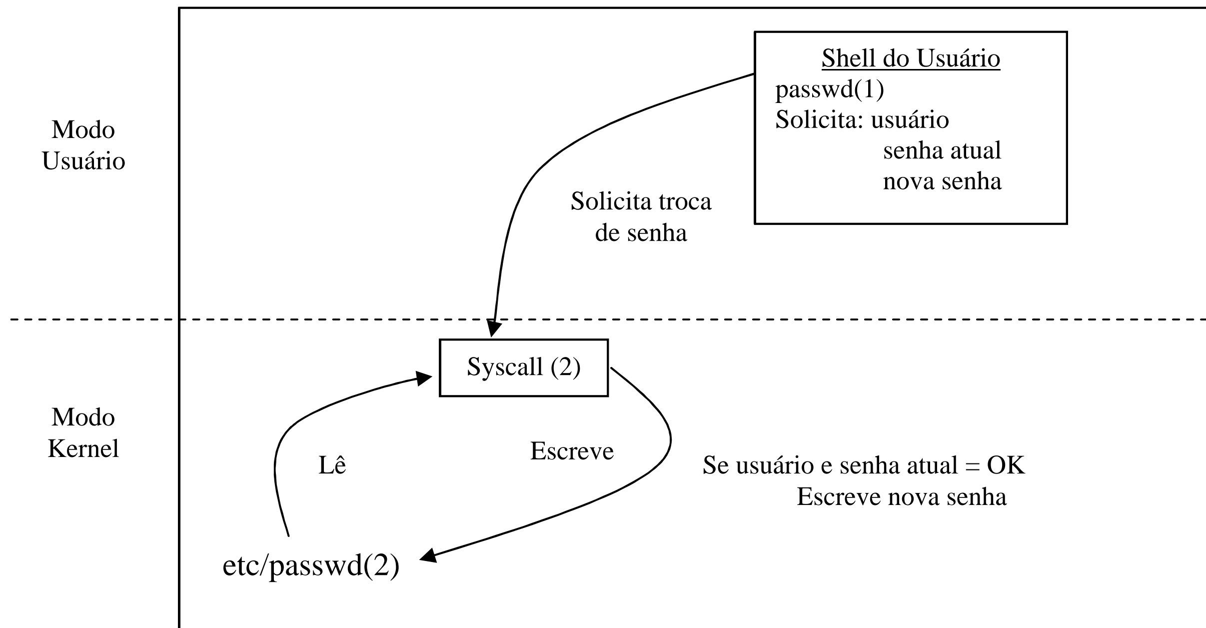

Figura 5.6. – Modelo Proposto para troca de senhas

5.7. Considerações sobre a proposta

A proposta esta baseada em 2 mecanismos que permitem a leitura e escrita não permitidos no modelo marca d´agua baixa do BIBA. Os mecanismos propostos não violam as regras do modelo e nem caracterizam canais cobertos (*covert channels*) [AMO94].

O mecanismo de transição entre nível permite a sujeitos de nível baixo de integridade armazenamento de informações em um repositório de nível alto de integridade. Fazendo-se uma analogia com o mundo físico é como permitir a um sujeito qualquer armazenar algo que julga importante (informação) em um cofre (repositório). Se considerado que esta opção está disponível apenas para arquivos texto e escrita no fim do arquivo (*append*), pode-se afirmar que a proposição não viola a regra NWU do modelo de integridade do BIBA. Pois se está permitindo a um sujeito guardar as informações que julga importante em local seguro, sem permitir ao mesmo que ganhe o controle do objeto.

A aplicabilidade do mecanismo proposto parece bastante restrita, mas não é, pois uma das maiores dificuldades da segurança computacional é obter registros de eventos no sistema nos quais se possa confiar (íntegros) e a proposta cumpre bem este papel. A separação dos registros de nível 1 e de nível 2 em dois arquivos, sem misturar os registros, reforça a colocação feita anteriormente de que não há violação da regra NWU, pois o repositório é de

nível 3 e o arquivo de nível 2, mas as informações constantes do arquivo */cont/log/messages1*, por exemplo, são apenas informações provenientes do nível 1.

O esquema proposto pode ser utilizado também para armazenar número de cartão de crédito para sistemas de compras *online*, onde o sujeito (comprador) é de nível 1 e a base de dados de nível 2. Neste caso, seria necessário utilizar também uma implementação do modelo de confidencialidade BLP, por exemplo, para evitar que o número dos cartões de crédito fosse roubada da base de dados.

O mecanismo codificado com separação de privilégios permite ganhar acesso a um objeto de nível de integridade alto a partir de um sujeito de nível baixo de integridade para a operação de escrita (atualização/*update*) quando o objeto em alguma de suas partes armazenando dados de determinado usuário. Ou seja, o objeto contém registro de diferentes usuários, os usuários não têm acesso ao objeto, mas podem ganhar acesso aos registros que lhe pertencem obtendo-se assim um controle de acesso fino ao objeto. Como a alteração (atualização) no objeto é solicitada por um sujeito de nível baixo de integridade, porem é efetivada por um código de nível alto de integridade, desta forma não há violação da regra de integridade NWU.

Em ambos os casos pode-se perceber que o comportamento discricionário acontece quando é permitido a um sujeito, proprietários de uma informação, por intermédio de um terceiro efetuar uma operação que o mesmo não conseguiria executar diretamente dado as regras da política multi-nível. Assim, sem violar as regras do modelo multi-nível um sujeito, seletivamente, pode fazer operações de leitura e escrita em dados que lhe pertencem e precisam fluir de um nível para o outro sem comprometer a proteção a arquivos executáveis que os modelos multi-nível oferecem, por exemplo, contra cavalos de tróia.

5.7.1. Benefícios da proposta

O modelo apresentado torna o sistema mais confiável e seguro, pois garante a distinção entre níveis de integridade fazendo com que o sistema funcione de forma que não sejam concedidos privilégios extras (sujeitos de confiança) a processo usuários. Desta forma, com a proposta não existirá mais o sujeito de confiança criado pelo LOMAC, onde um sujeito de alto nível de integridade poderia ler um objeto de baixo nível de integridade sem rebaixar o

seu nível, e com isto, caso um intruso ganhasse os privilégios do sujeito de confiança, poderia comprometer o sistema como um todo. Porém, é importante observar que para o caso dos demais programas executáveis que não o *syslogd* e *passwd*, o comportamento do LOMAC permanece o mesmo.

Torna-se relevante o beneficio de que um sujeito de baixo nível de integridade tenha suas informações gravadas em um repositório de alto nível de integridade, onde somente este sujeito, terá permissões para ler e escrever as informações que lhe são pertinentes. Como exemplo se pode destacar troca de senha de um sujeito de baixo nível de integridade.

5.7.2. Limitações da proposta

O modelo proposto pode sofrer ataques de negação de serviços no objeto de transição de nível (*/dev/ddlomac*). A negação de serviço neste caso, não foi preocupação desta proposta, porém como comentado no fim da seção 5.5 o sistema de *log* deverá continuar operando.

Como o comportamento do LOMAC para programas executáveis (binários) não é alterado a exceção do *passwd* e *syslogd* a proposta tem a limitação, pois não se pode distinguir entre um programa que leia dados de nível baixo de integridade e que ainda está funcionando corretamente de um que leu dados do nível baixo de integridade e foi corrompido.

Como as versões do LOMAC para Linux somente foram implementadas corretamente até a versão do kernel 2.2, não se tem uma solução atualizada para o caso. Porém, pode-se tornar viável está proposta para o FREEBSD que possui o LOMAC em sua atual versão.

5.8. Conclusão

Este capítulo descreveu uma proposta para aumentar a segurança de sistemas operacionais através da adição do LOMAC ao Linux, tornando o mesmo livre de exceções que violavam as regras do modelo BIBA. Foram acrescentados novos mecanismos ao Linux para que o mesmo se torne mais compatível com a utilização do LOMAC. Para eliminar os sujeitos de confiança foi alterado o comportamento do LOMAC e criado no código do mesmo um outro nível de integridade (nível 3), que tem a finalidade de ser um repositório para armazenar informações de baixo (nível 1) e alto nível de integridade (nível 2), desta forma se

tem uma maior interação entre os diferentes níveis de integridade sem ferir as regras impostas pelo modelo de integridade BIBA. A proposta em um contexto geral prevê um novo modelo de comportamento entre diferentes níveis de integridade e a criação de um modelo para que sujeitos de baixo nível de integridade possam alterar informações que lhe pertencem, mas estão em um nível de integridade superior ao seu. Com estas duas propostas; caracteriza-se que está se provendo um comportamento discricionário ao modelo de integridade BIBA de marca d´água baixa do sujeito.

Capítulo 6

Implementação e Resultados

Para realizar os testes do modelo proposto no capítulo anterior, foi implementado um protótipo. Este protótipo foi implementado em plataforma Linux distribuição RedHat 6.2 (kernel 2.2.14), utilizando a versão do LOMAC 1.1.2. A implementação abrange a criação de um dispositivo de transição baseado em OSE e um serviço de leitura e escrita de registros de eventos. A alteração do código do LOMAC se fez necessária para que o mesmo crie o nível de integridade (3) utilizado para ser o repositório de registros de eventos do sistema e do kernel. Para obter esta funcionalidade foi alterado o arquivo de configuração do *syslog*.

A implementação em questão, a alteração dos códigos citados e também os resultados obtidos com a utilização do protótipo, serão apresentadas no decorrer deste capítulo.

6.1. Alteração do código do LOMAC

A alteração do código do LOMAC se fez necessária devido a três necessidades para criação do modelo proposto. A primeira necessidade foi à criação do nível de integridade (3) com o intuito de ser o repositório dos registros de sistema e de kernel, os quais possuem baixo e nível alto de integridade respectivamente. O nível de integridade (3) que será o repositório de registros de eventos do kernel e do sistema, terá dois arquivos para armazenamento de eventos (messages1 e messages2).

A segunda necessidade é a alteração e criação de novos arquivos dentro da implementação do LOMAC, este passo se torna necessário devido a criação do terceiro nível de integridade criado no LOMAC. Esta alteração tem a função de estabelecer o nível de

integridade 3 (repositório de informações). O diretório que será o repositório possuirá informações de baixo e alto nível de integridade. Os registros de eventos do sistema serão gravados pelo serviço KTlog no arquivo messages1 e os registros de eventos do kernel serão gravados pelo serviço klogd no arquivo messages2, ambos arquivos armazenados dentro do repositório.

A terceira e última modificação é a retirada do *daemon syslogd* da opção de exceções criadas pelo LOMAC (sujeitos de confiança), onde um sujeito de alto nível de integridade pode ler e/ou escrever em um objeto de baixo nível de integridade sem que seu nível de integridade seja rebaixado. A retirada do *daemon syslogd* desta exceção, torna a utilização do LOMAC mais confiável, pois assim, não serão violadas as regras impostas pelo modelo de integridade BIBA. Outro ponto positivo, é que deste modo, o arquivo de registros do sistema estará integro em um nível superior de integridade, não ocorrendo a possibilidade de um sujeito de baixo nível de integridade, ganhar controle do *syslogd* (nível alto de integridade) e comprometer o sistema.

6.2. Alteração do arquivo de configuração do *Syslog*

A alteração do arquivo de configuração do *syslog* (*/etc/syslog.conf*) se faz necessária devido a alteração do código do LOMAC para criação do terceiro nível de integridade e também para avaliação da proposta.

As mudanças que ocorreram no arquivo de configuração se fazem necessárias, pois os *daemons syslogd* e *klogd* utilizam este arquivo para obterem informações sobre quais são os parâmetros fornecidos para os mesmos gerarem os *logs* e também para saberem em quais arquivos irão gravar tais informações. As mudanças realizadas foram as seguintes:

- ***Klogd:*** por padrão, na definição de configuração de *logs* do Linux, o *daemon klogd* gera as informações do kernel e as grava em um console de saída padrão (*/dev/console*); a modificação realizada faz com que o *klogd*, que está funcionando em nível alto de integridade de acordo com o LOMAC, armazene as informações do kernel no arquivo criado dentro do repositório de nível (3);
- ***Syslogd***: por padrão, na definição de configuração de *logs* do Linux, o *daemon syslgod* lê as informações de um *pipe* padrão de *log* do sistema (*/dev/log*) e as grava em um

arquivo (*/var/log/messages*). A modificação realizada no arquivo de configuração faz com que o *daemon syslogd* faça a gravação dos registros em um arquivo criado pela implementação proposta (*/dev/ddLOMAC*) que será o dispositivo baseado em OSE.

6.3. Implementação do dispositivo baseado em OSE

Para realizar a implementação do dispositivo baseado em OSE que terá a finalidade de ser um objeto de transição entre o baixo e o alto nível de integridade, foi utilizada a implementação de dispositivo de caractere que obedece os critérios estabelecidos para ser um objeto sem-estado ou objeto não persistente.

O dispositivo baseado em OSE será a transição entre os serviços *syslogd* baixo nível de integridade e KTlog (serviço proposto) alto nível de integridade. A cada invocação realizada pelos serviços citados, o objeto ficará sem as informações que já foram consumidas; desta forma, manterá as informações necessárias entre uma transição e outra. Um outro ponto a ser destacado com a utilização do objeto de transição, é que se caso algum intruso invada o sistema e tente apagar as informações contidas no dispositivo OSE, o mesmo não obterá sucesso, já que estas informações já foram consumidas pelo serviço KTlog e estão gravadas no repositório em nível 3 de integridade.

6.4. Implementação do KTlog

Como descrito no capítulo anterior, o KTlog terá a função de realizar a leitura do objeto de transição e realizar a gravação da informação lida no repositório de registros. O KTlog foi implementado como um serviço que irá funcionar em modo kernel. Deste modo, foi utilizado o conceito de kernel *threads* para a criação do serviço e está demonstrado na Figura 6.1. Como descrito no capítulo anterior, a leitura e escrita das informações realizadas pelo serviço KTlog foi implementada utilizando chamadas de sistema.

O KTlog foi implementado como um LKM (*Loadable Kernel Module*) para que o serviço seja acionado antes da implementação do LOMAC, pois devido ao funcionamento do LOMAC, onde o mesmo intercepta as chamadas do sistema (descrito no capítulo 4), o KTlog tem que carregar a tabela de chamadas de sistema antes que o LOMAC, caso contrário não irá funcionar.

```
                          /* Kernel Versão 2.2.14 */

extern void *sys_call_table[];
asmlinkage ssize_t (*sendfile)(int  out_fd,  int  in_fd, off_t *offset,
size_t count);
asmlinkage int (*fstat)(int filedes, struct stat *buf);
asmlinkage int (*open)( const char *filename, int flags);
asmlinkage int (*open1)( const char *filename, int flags, mode_t mode);
asmlinkage int (*close)(int);


int init_module(void)
{
      int i;

      /*Chamadas de sistema utilizadas pelo KTlog */
      open = sys_call_table[__NR_open];
      open1 = sys_call_table[__NR_open];
      close = sys_call_table[__NR_close];
      sendfile = sys_call_table[__NR_sendfile];
      fstat = sys_call_table[__NR_fstat];

        /* Criação de Novas Threads */
      for (i=0; i <NTHREADS; i++)
        launch_thread((int (*)(void *))example_thread, &example[i]);

        return(0);
}
```

Figura 6.1. – Chamadas de sistema utilizadas pelo KTlog

Um detalhe importante que vale ser comentado é o funcionamento do kernel *thread* implementado, onde o mesmo necessita de uma função responsável pela geração de um *daemon* em *background* para poder funcionar. É a função *daemonize*, que em kernel versão 2.2.14 tem que ser implementada dentro do código utilizado e caso esteja-se utilizando kernel 2.4 ou superior, basta fazer a chamada da função e criar a ligação com a biblioteca necessária para poder utilizá-la. Outras funções utilizadas pelo KTlog são responsáveis por gerenciar a kernel *thread* e também a utilização de semáforos do kernel. A funções utilizadas são as seguintes: *start_kthread* (criar uma nova kernel thread), *stop_kthread* (parar um kernel thread), *init_ kthread* (configurar o ambiente para chamar a nova kernel thread) e *exit_kthread* (necessário para terminar uma kernel thread e sair).

Como citado anteriormente, para realização de leitura e escrita o KTlog foi implementado utilizando chamadas do sistema. Devido ao problema enfrentado com a interceptação das chamadas do sistema, foi necessário fazer uma análise e verificar quais chamadas do sistema são interceptadas pelo LOMAC. Em princípio, a implementação de

leitura e escrita era realizada pelas chamadas *read* e *write*, mas devido a problemas enfrentados durante testes de implementação junto ao LOMAC, decidiu-se por não utilizar estas chamadas de sistema na implementação do KTlog; outro motivo para isto advém do fato de o LOMAC interceptar estas chamadas para poder realizar um controle melhor do sistema.

Para realizar a implementação final do KTlog foram utilizadas três chamadas de sistema: *open*, *close* e *sendfile*. As duas primeiras chamadas citadas são interceptadas pelo LOMAC, mas não geram problemas ao serviço KTlog; já a utilização da *sendfile* (não interceptada pelo LOMAC) como visto na função utilizada para implementação do protótipo na Figura 6.2, torna o serviço mais confiável, devido ao fato que a mesma não utiliza a implementação de *buffer* interno para realizar a troca de informações entre o objeto de transição e o repositório de dados. Desta forma, como o serviço está funcionando em modo kernel a utilização de um *buffer* interno pode ser considerado perigoso para o sistema como um todo [BIS03, SIL02].

```
void rw_arq(void)
{

      int read_fd, write_fd;

      struct stat stat_buf;

      off_t offset=0;

      mm_segment_t old_fs = get_fs();

      set_fs(KERNEL_DS);


      read_fd = open("/var/log/messages", O_RDONLY);

      fstat (read_fd, &stat_buf);

      write_fd = open1("/cont/log/messages1", O_WRONLY |
      O_TRUNC, stat_buf.st_mode);

      sendfile(write_fd, read_fd, &offset, stat_buf.st_size);

      close(read_fd);

      close(write_fd);

      set_fs(old_fs);
}
```

Figura 6.2. – Função de utilização da chamada de sistema sendfile

A utilização de serviços funcionando em *background* gera um custo de utilização de CPU (*Central Processing Unit*), já que o serviço funciona em um laço de repetição (até que seja solicitada a sua parada), isto é, de tempos em tempos o kernel *thread* realiza a verificação do objeto de transição, a fim de verificar se o mesmo possui informações para serem coletadas. Devido ao fato da utilização deste laço de repetição, é utilizado um "temporizador" para o serviço KTlog, determinando de quanto em quanto tempo será realizada a verificação necessária no objeto de transição. A implementação da kernel *thread* utiliza o "temporizador", para determinar o tempo que o serviço ficará parado até voltar a realizar a sua verificação; desta maneira quanto menor o tempo que o mesmo ficar esperando, maior será o consumo de ciclos de CPU. O código utilizado pelo kernel *thread* para realizar a verificação do objeto de transição é demonstrado na Figura 6.3.

Uma implementação mais elaborada poderia ser realizada utilizando-se semáforos. A implementação usando semáforo não foi desenvolvida neste trabalho, pois o consumo de ciclos de CPU utilizando uma verificação constante (caso utilizado), não é um consumo que afeta o desempenho do sistema como um todo. Outro motivo para não elaboração com utilização de semáforos se deve a restrições temporais.

```
void example_thread(my_threads *thread)
{

      /* Configuração do ambiente */

        setup_thread(thread);
        printk("Iniciando kernel thread ...\n");


      /* Ciclo de funcionamento da Thread*/

        do{

            interruptible_sleep_on_timeout(&thread->queue, 2*HZ);

            mb();

            rw_arq();

        }while(!thread->terminate);

        leave_thread(thread);
}
```

Figura 6.3. – Função de utilização da Kernel Thread

6.5. Avaliação de desempenho do modelo proposto

Para realizar a avaliação do protótipo, foi utilizado um equipamento com a seguinte configuração: CPU Athlon XP 2.1 (1.7 GHZ) – 512 MB DDR-PC333 – HD IDE 20 GB ATA 100. O sistema operacional utilizado foi o Linux distribuição RedHat 6.2, kernel 2.2.14. A utilização do RedHat 6.2 foi devido a implementação do LOMAC suportar sem problemas até esta versão.

6.5.1. Desempenho do KTlog *read/write*

Para realização dos testes de desempenho do serviço KTlog, foram realizados testes; onde se levou em consideração a utilização de diferentes chamadas de sistema na implementação do serviço proposto. A tabela 6.1 demonstra a relação entre o tempo que a kernel *thread* irá ficar esperando entre uma repetição de verificação e outra e a utilização das chamadas *read/write* para solução do problema de leitura e escrita.

Chamada de Sistema	Tempo (s)	Consumo de CPU (%)
Read / Write	0,001	80,52
Read / Write	0,01	83,23
Read / Write	0,1	33,53
Read / Write	1	4,18
Read / Write	2	2,39
Read / Write	4	1,30
Read / Write	8	0,40

Tabela 6.1. – Tempo e Consumo do KTlog Read/Write

6.5.2. Desempenho do KTlog sendfile

Para realização do teste de desempenho do serviço proposto KTlog; onde se levou em consideração a chamada de sistema *sendfile*, foram utilizados os mesmos critérios das chamadas de sistema *read/write*. A tabela 6.2 mostra que a solução do problema, utilizando a chamada de sistema *sendfile* obtém um melhor desempenho que a solução anterior.

Chamada de Sistema	Tempo (s)	Consumo de CPU (%)
Sendfile	0,001	0,60
Sendfile	0,01	0,20
Sendfile	0,1	0
Sendfile	1	0
Sendfile	2	0
Sendfile	4	0
Sendfile	8	0

Tabela 6.2 – Tempo e Consumo do KTlog Sendfile

Como a utilização do serviço proposto será para ter seu funcionamento no espaço do kernel, foi optado pela implementação que utiliza a chamada de sistema *sendfile*, pois a mesma não utiliza *buffer* interno para fazer a troca entre informação lida e escrita em diferentes arquivos, assim, evitando o problema de *buffer-overflow* que poderia acontecer. Como visto na tabela 6.2, o consumo de processamento do serviço proposto é adequado para uma solução deste porte, pois funciona nos moldes do serviço nativo, e outro detalhe é que fornece maior proteção contra intrusos porque está funcionando em modo kernel obtendo do LOMAC um alto nível de integridade.

6.5.3. Avaliação de desempenho do protótipo

Para avaliação de desempenho do protótipo, foram realizados testes executando os comandos *find, ps* e *make*. Os testes foram realizados utilizando 3 situações distintas para cada comando. Desta forma, foi verificado que a utilização do protótipo, não impacta no desempenho do Linux com LOMAC, já que o impacto maior de desempenho se deve a utilização do LOMAC, o qual tem um impacto de 5% do desempenho na sua utilização com o Linux [FRA01], como demonstrado na tabela 6.3.

Comando	Linux Nativo	Linux com LOMAC	Linux com LOMAC e LKM Proposto
find –depth –name *.h	0,28 seg	0,45 seg	0,45 seg
ps –ef	0,02 seg	0,02 seg	0,03 seg
make /root/LOMAC	0,20 seg	0,24 seg	0,24 seg

Tabela 6.3 – Avaliação Desempenho do Protótipo

Os resultados obtidos na tabela 6.3 se deram pela média extraída da execução dos comandos. Cada comando foi executado 50 vezes obedecendo ao mesmo padrão em cada execução. Através dos totais obtidos nas execuções, foram extraídas as médias para geração da tabela acima.

6.5.4. Testes com a utilização de *exploit*

Para realização do teste utilizando um *exploit*, foi verificado o funcionamento do *exploit imapd* com a execução no Linux Nativo e também a utilização do Linux com o LOMAC.

No primeiro teste realizado com o Linux Nativo, depois de executado o *exploit*, conseguiu-se uma *shell* do sistema operacional tornando-o suscetível ao comprometimento.

No segundo teste realizado com o Linux rodando o LOMAC modificado mais o protótipo, também se obteve uma *shell* do sistema, mas devido a ação do LOMAC, a *shell* possui um baixo nível de integridade e o intruso que solicitou tal operação também possui baixo nível de integridade, logo não há perigo potencial de comprometimento do sistema como um todo.

A *shell* obtida pelo intruso só permite que este cause danos aos objetos que estiverem em baixo nível de integridade, não podendo afetar a parte do sistema que está em alto nível de integridade.

Como a exceção criada pelo modelo padrão do LOMAC foi revista pelo protótipo em questão, isso limita a ação do intruso. Deste modo, o intruso não tem a possibilidade de tentar ganhar o alto nível de integridade, pois o *daemon syslogd* está rodando em baixo nível de integridade e a possibilidade do intruso apagar seus rastros do arquivo de registro de eventos do sistema, inexiste. Desta forma o administrador do sistema pode verificar tal invasão nos arquivos de registros do sistema e/ou do kernel.

6.5.5. Testes de parada do serviço de registro

Para realização deste teste foi utilizado o Linux com LOMAC modificado mais o protótipo. Foi criado um usuário com permissões restritas e se verificou que o usuário criado

possui um baixo nível de integridade mas com a possibilidade de realizar algumas ações na parte do sistema que está em nível baixo de integridade.

O primeiro teste realizado foi parar o serviço de registro *syslogd (kill "syslogd pid")*, ação que obteve sucesso e o arquivo de armazenamento não pode ser atualizado, mas estas ações que foram realizadas ficaram gravadas no arquivo de armazenamento do serviço de registro do kernel *klogd* e puderam ser verificadas sem problemas pelo administrador. Foi realizada a ação de parar o serviço *klogd* (*kill "klogd pid"*), mas esta ação não obteve sucesso devido ao fato que o serviço *klogd* estava em nível alto de integridade.

O segundo teste realizado foi a tentativa de apagar e/ou alterar o arquivo gerado pelo serviço proposto KTlog, mas esta ação também não obteve sucesso, devido ao fato do arquivo de armazenamento gravado pelo KTlog estar em um repositório de informações com nível de integridade (3).

6.6. Análise dos resultados obtidos

Com os testes realizados, foi verificado que a solução do LOMAC gerando dois níveis de integridade no sistema operacional mais o nível 3 com a finalidade de guardar informações de baixo e alto nível de integridade são importantes para se ter um ambiente seguro.

Os testes realizados para verificações de desempenho do serviço proposto deixaram evidente que o mesmo é viável para solução do problema de exceção do LOMAC. A funcionalidade do serviço proposto KTlog com a utilização da chamada de sistema *sendfile* o torna seguro, devido à não utilização de buffer para troca de informações. Outro ponto fundamental é a utilização do conceito de objeto-sem-estado para ser o objeto de transição entre os níveis de integridade, desta forma não violando o modelo de integridade BIBA.

6.7. Conclusão

Este capítulo descreveu as partes do modelo proposto, definindo como as mesmas foram implementadas e também as alterações que se fizeram necessárias no código do LOMAC e no arquivo de configuração do serviço de registro do Linux. Foram mostrados os

testes realizados para verificação da viabilidade da proposta em termos de desempenho e também sua funcionalidade dentro do sistema operacional.

Capítulo 7

Conclusão

O desenvolvimento deste trabalho apresenta uma proposta de extensões ao mecanismo de controle de integridade multinível (LOMAC), tornando-o mais compatível com o modelo de integridade multinível do BIBA. A proposta oferece uma alternativa aos relaxamentos feitos nas regras de política multinível da implementação do LOMAC no Linux.

Uma das extensões propostas neste trabalho permite que um sujeito de baixo nível de integridade faça a escrita seqüencial de registros (string de caracteres) - com o mesmo nível de integridade - em um repositório com um nível superior de integridade. Outra extensão proposta permite que um sujeito de baixo nível de integridade realize a atualização (escrita) de registros – de mesmo nível de integridade – em um repositório de alto nível de integridade.

A implementação do protótipo mostrou de forma prática a funcionalidade do modelo proposto, onde se tem a garantia dos registros de eventos do sistema e do kernel, que são gerados por dispositivos independentes sem a intervenção de outros processos. Caso ocorra a desativação de processos que estejam em funcionamento, esta desativação irá gerar um registro de evento, que será fornecido tanto ao kernel quanto ao sistema. Deste modo, se for desativado o serviço de registro do sistema que está em um nível baixo de integridade, este evento será registrado pelo serviço de eventos do kernel, e esta desativação será percebida pelo administrador do sistema. Desta forma, as informações dos eventos do sistema não serão atualizadas, mas permanecerão seguras, pois estão gravadas em um arquivo que está em nível alto de integridade. Porém, o serviço *klgod* continuará gravando os eventos do kernel, através do repositório criado pela implementação, garantindo assim a segurança do sistema.

Estão sendo realizadas melhorias no mecanismo apresentado, e se pretende torná-lo viável para utilização em versões mais recentes do Linux. Outro trabalho que está sendo estudado é a implementação de um ambiente operacional com a utilização do conceito de virtualização do sistema operacional, onde se pretende trabalhar com mecanismos de controle de acesso obrigatório no modo real e modo virtual do ambiente operacional. E por último, está sendo estudada a viabilidade da implementação do modelo proposto no capítulo 5 para solução do problema da troca de senhas por usuários comuns.

As principais contribuições deste trabalho foram:

- A utilização de um dispositivo de caractere para assumir o papel do objeto sem estado. A adoção deste objeto de transição se apresentou como a melhor solução em relação a utilização de *pipe* ou *soquete*, pois os mesmos permitem o controle do fluxo de informação na sua concepção;
- A elaboração de um repositório de informações criado junto ao LOMAC, tendo como propósito ser um repositório de informações de baixo e alto nível de integridade armazenados em separado. Deste modo, o repositório criado, armazena diferentes informações sem misturá-las, obedecendo ao comportamento de um cofre;
- Ajuste do nível de integridade do *daemon syslogd* (alterado de alto nível de integridade para baixo nível de integridade), evitando que um ataque bem sucedido ao *daemon syslogd* (rodando como processo usuário) pudesse comprometer as informações com alto nível de integridade;
- Maior proteção dos registros do sistema e do kernel. Caso seja lançado um ataque do tipo DoS sobre o s*yslogd* (baixo nível de integridade), o *klogd* (alto nível de integridade) continuará registrando as informações dos eventos do kernel e portanto não serão perdidas as informações importantes, pois estas informações geradas pelo *klogd* estarão em um repositório de nível superior de integridade e não serão afetadas. As informações geradas pelo *syslogd* até o momento do ataque, também estarão inalteradas, devido ao fato que estarão no repositório em questão;
- Criação de um modelo para que um sujeito com baixo nível de integridade possa realizar alterações em informações (que sejam pertinentes ao mesmo) que possuam um alto nível de integridade. Fazendo-se a utilização da técnica de programação de separação de privilégios;

- Através dos testes realizados com a implementação do serviço de leitura e escrita entre o objeto de transição e o repositório em nível superior de integridade demonstraram que a utilização da chamada de sistema *sendfile* obteve um melhor desempenho do que a utilização das chamadas de sistema *read* e *write*. Além do melhor desempenho apresentado, a utilização da chamada de sistema *sendfile* é menos suscetível a ataques de *buffe -overflow*, pois a mesma realiza a troca direta (utilização apenas do descritor de arquivos) de informação entre arquivo destino e arquivo origem, sem a necessidade de utilizar um buffer interno de comunicação entre a leitura e escrita. Ao contrário, as chamadas de sistema *read* e *write* utilizam um *buffer* para realizar a troca de informações, tornando o processo mais lento;
- A implementação do modelo em forma de LKM e funcionando em modo kernel. Desta forma, o serviço proposto grava as chamadas de sistema em memória antes que o LOMAC seja acionado, pois o LOMAC quando está sendo acionado intercepta as chamadas de sistema para realizar um controle de integridade mais efetivo. O funcionamento do serviço proposto em modo kernel o torna mais aplicável ao modelo, pois o mesmo estará funcionando em um alto nível de integridade e também devido às interceptações das chamadas de sistema realizadas pelo LOMAC.
- O modelo proposto provê que sujeitos que estão em um baixo nível de integridade possam alterar (escrever) os objetos pertencentes aos mesmos e que estão em um alto nível de integridade; O modelo provê um comportamento discricionário ao modelo obrigatório de integridade BIBA, pois torna viável que sujeitos que estejam em um baixo nível de integridade tenham acesso aos objetos pertencentes aos mesmos, através de sua descrição. Esta operação se torna possível, pois a comunicação multi-nível entre sujeitos e objetos é intermediada por um outro dispositivo (baseado em separação de privilégios e/ou objeto sem estado). Por exemplo: No mundo físico, se um sujeito tem dinheiro guardado em um banco e queira resgatar este dinheiro, ele necessita requisitar ao gerente do banco ou a pessoa responsável para proceder tal operação. Desta forma, o dinheiro guardado no banco é de propriedade de tal sujeito, mas o mesmo não tem acesso direto ao que lhe pertence.

Com a apresentação dos itens acima se chega à conclusão que a proposta apresentada neste trabalho é viável e demonstra vários benefícios para modificação e construção de

mecanismos de controle de acesso baseado no modelo obrigatório e também a sua utilização em conjunto com o modelo discricionário.

REFERÊNCIAS BIBLIOGRÁFICAS

[AMO94] AMOROSO, E. G. *Fundamentals of Computer Security Technology*. Computer Security. Prentice Hall. 1994.

[AND72] ANDERSON, J. P. *Computer Security Technology Planning Study*. Report ESD-TR-73- 51.Electronic Systems Division. 1972.

[BEL76] BELL, D. E. ; LAPADULA, L. J. . *Secure Computer System: Unified Exposition and Multics Interpretation*. MITRE. 1976.

[BER02] BERNASCHI, M; GABRIELLI, E; MANCINI, L. V. *Remus: A security enhanced operating system*. ACM Transactions on Information and System Security. 2002.

[BIB77] BIBA; K. J. . *Integrity considerations for secure computer systems*. MITRE MTR-3153. 1977.

[BIS03] BISHOP, M. *Computer Security: Art and Science*. Addison Wesley. 2003.

[BOE93] BOER, B.; BOSSELAERS, A. *Collisions for the Compression Function of MD5*. EUROCRYPT. 1993.

[BOR01] BORCHARDT, M. ; MAZIERO, C. A. . *Verificação da integridade de arquivos no kernel do sistema operacional*. Workshop em Segurança de Sistemas Computacionais (SBRC). 2001.

[BUR95] BURROWS, J. H. *Secure Hash Standard*. FIPS PUB 180-1, Washington D.C., 1995. Disponível em: < http://www.itl.nist.gov/fipspubs/fip180-1.htm >. Acesso em: maio 2005.

[CLA87] CLARK, D. D. ; WILSON, D. R. . *A comparison of commercial and military computer security policies*. In Proceedings of the IEEE Symposium on Security and Privacy. 1987.

[DEN82] DENNING, Dorothy. *Criptography and data security*. Addison-Wesley. 1982.

[FRA99] FRASER, T. ; BADGER, LEE. ; Feldman, M. . *Hardening COTS Software with Generic Software Wrappers.* IEEE Symposium on Security and Privacy. 1999.

[FRA00] FRASER, T. *LOMAC*: *Low Water-Mark Integrity Protection for COTS Environments*. IEEE Symposium on Security and Privacy. 2000.

[FRA01] FRASER, T. *LOMAC: MAC You Can Live With*. USENIX Technical Conference, Boston, USA. 2001.

[GEH82] GEHRINGER, E. F. . *Capability Architectures and Small Objects*. UMI Press. 1982.

[HAN01] HANSEN, T. ; WOLLMAN, U. S. . *Secure Hash Algorithm 1 (SHA1)*. RFC 3174. 2001. Disponível em: < www.ietf.org/rfc/rfc3174.txt>. Acesso em: julho de 2005.

[KIM 95] KIM, G. H.; SPAFFORD, E. H. *The Design and Implementation of Tripwire: A File System Integrity Checker*. Purdue University. 1993.

[LAM71] LAMPSON, B. W. *Protection*. Proc. 5th Princeton Conf. on Information Sciences and Systems, Princeton. 1971.

[LAN81] LANDWEHR, C. E. *Formal models for computer security*. Computing surveys. 1981.

[LAN01] LANDWEHR, C. E.. *Computer security*. IJIS . 2001.

[LAU04] LAUREANO, M. A. P. ; MAZIERO, C. A. ; JAMHOUR, E. . *Proteção de detectores de intrusão através de máquinas virtuais*. Workshop em Segurança de Sistemas Computacionais (SBRC). 2004.

[LET05] LEHTI, R.; VIROLAINEN, P. *AIDE – Advanced Intrusion Detection Environment. 2001*. Disponível em: <http://www.cs.tut.fi/~rammer/aide.html>. Acesso em: maio 2005.

[LOS98] LOSCOCCO, P. A.; SMALLEY, S. D. *Meeting critical security objectives with security-enhanced linux*. 1998.

[MEN96] MENEZES, A. J. *Handbook of Applied Cryptography*. Florida: CRC Press, 1996.

[MIT97] MITCHEM, T. ; LU, R. ; O'BRIEN, R. . *Using kernel hypervisors to secure applications*. 13th Annual Computer Security Applications Conference. 1997.

[NIC97] NICOMETTE, V; DESWARTE, Y. *An Authorization Scheme For Distributed Object Systems.* IEEE Symposium on Security and Privacy. 1997.

[ODE04] ODESSA, U. I.. *VXE - Virtual executing environment* – 2004. Disponível em: <http://vxe.quercitron.com/Overview/overview.html>. Acesso em: junho de 2005.

[PRO03] PROVOS, N. ; FRIEDL , M.; HONEYMAN, P. *Preventing Privilege Escalation*. USENIX Security Symposium. 2003.

[PAY04] PAYNE, Christian. *Enhanced Security Models for Operating Systems: A Cryptographic Approach*. IEEE Computer Software and Applications Conference. 2004.

[RIV92] RIVEST, R. *The MD4 Message-Digest Algorithm: RFC 1320*. 1992. Disponível em: < http://www.ietf.org/rfc/rfc1320.txt >. Acesso em: julho 2005.

[RIV02] RIVEST, R. *The MD5 message-digest algorithm*. RFC 1321. 1992. Disponível em: <http://www.ietf.org/rfc/rfc1321.txt>. Acesso em: maio 2005.

[SAL75] SALTZER, J. H.; SCHROEDER, M. D. *The Protection of Information in Computer Systems*. IEEE Cryptography and Data Security. 1982.

[SAN00] SANDHU, R., Ferraiolo, D. and Kuhn, R. *The NIST Model for Role-Based Access Control: Towards A Unified Standard*. ACM workshop. 2000.

[SCH96] SCHNEIER, B. *Applied Cryptography*. New York: John Wiley & Sons, 2ª edição. 1996.

[SHI00] SHIRLEY, R. *Internet Security Glossary*. IETF, RFC 2828. 2000. Disponível em: <http://www.ietf.org/rfc/rfc2828.txt>. Acesso em: julho 2005.

[SIL02] SILBERSCHATZ, A. *Sistemas operacionais: conceitos e aplicações*. Campus – 2002.

[TAM05] TAMBORIM, A. L. *Introdução a LIDS*. 2005. Disponível em: <http://www.y2h4ck.hpg.ig.com.br/lids.htm >. Acesso em: maio 2005.

[TAN03] TANENBAUM, A. S.. *Sistemas Operacionais Modernos*. Prentice Hall - 2ª Edição. 2003.

[WAG99] WAGNER, D. A. *Janus: An Approach for Confinement of Untrusted Applications*. CSD. 1999.

[WRI02] WRIGHT, C.; COWAN, C.; MORRIS, J.; SMALLEY, S.; KROAH-HARTMAN, G. *LSM*: *General Security Support for the Linux Kernel.* USENIX Security Symposium. 2002.

[ZEL97] ZELEM, M. ; PIKULA, M. ; OCKAJAK, M. . *Medusa DS9 Security System*. – 1997. Disponível em: <http://medusa.fornax.sk/English/project.shtml>. Acesso em: junho 2005.

This document was created with Win2PDF available at http://www.daneprairie.com.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.