UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ARNALDO GOMES DO AMARAL

UM MECANISMO DE PROTEÇÃO DE ARQUIVOS POR SENHAS EMBUTIDO NO NÚCLEO DO SISTEMA OPERACIONAL LINUX

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação

Prof. Thadeu Botteri Corso
Orientador

Florianópolis
Julho - 2002

UM MECANISMO DE PROTEÇÃO DE ARQUIVOS POR SENHAS EMBUTIDO NO NÚCLEO DO SISTEMA OPERACIONAL LINUX

ARNALDO GOMES DO AMARAL

Esta Dissertação foi julgada adequada para obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistema de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____

Prof. Fernando Alvaro Ostuni Gauthier, Dr.

Banca Examinadora

_____

Prof. Thadeu Botteri Corso, Dr. (Orientador)

_____

Prof. Luis Fernando Friedrich, Dr.

_____

Prof. José Mazzucco Júnior, Dr.

Dedico este trabalho a cinco mulheres que participam da minha vida, Sandra (esposa), Francielly, Francinny e Francianny (filhas) e Antonia (mãe).

AGRADECIMENTO

Muitas pessoas participaram direta ou indiretamente na elaboração deste trabalho, portanto não poderia deixar de manifestar aqui minha profunda gratidão por todas elas. Agradeço em especial:

- Ao Senhor Deus todo poderoso, Rei supremo, o qual controla e dirige minha vida, por esta oportunidade de crescimento.
- Ao meu orientador Prof. Thadeu, pela dedicação, ensinamento e compreensão, sem o qual seria difícil completar esta empreitada.
- Aos meus pais, José (in memoriam) e Antonia, pela educação recebida, através da qual posso manter um convívio de amizade e com isto melhorar cada vez mais como ser humano e, assim, atingir meus objetivos futuros.
- A Dona Alice (madrinha) pelos cuidados e preocupação, possibilitando hoje estar aqui finalizando este trabalho.
- À minha esposa Sandra, pelo amor, carinho e dedicação. Ponto de equilíbrio da minha vida, mostrando-me o lado humano quando insisto demais no técnico. E indicando caminhos simples quando estou procurando soluções complicadas para resolver problemas.
- Às filhas Francielly, Francinny e Francianny, que participaram deste trabalho deixando sua contribuição privando de minha companhia.
- Aos professores e amigos, Augusto Carlos Castro dos Santos, Maria Aparecida e Clovis, Douglas V. Xavier, Nelson e Armando Posseti, Régio e Fátima Gimenes, José Filho e Sonia Oliveira pelo apoio, carinho, dedicação e palavras de incentivo nos momentos mais difíceis da vida acadêmica e profissional.
- A Universidade Paranaense – UNIPAR, nas pessoas do Dr. Cândido e Prof.ª Neiva, pela oportunidade de fazer este curso.

- Aos companheiros do Grupo de Comunidade, pelo apoio, preocupação e orações nos momentos de dificuldade e decisão que passamos pela vida.
- Ao colega Aurio Dreer pela ajuda e esclarecimento de dúvidas.
- Aos monitores do Laboratório de Informática da Unipar.
- Ao colega Silvanei, companheiro de várias jornadas, falta agora terminar seus estudos!
- Ao Prof. José Tereziano, pela ajuda na correção do texto.
- Aos companheiros e amigos Altevir e Dorotéia, pelo apoio dedicado mesmo à distância.

SUMÁRIO

LISTAS DE FIGURAS

LISTAS DE TABELAS

RESUMO

O objetivo deste trabalho é inserir um sistema de proteção de senha em arquivos no sistema operacional Linux e que a mesma seja gerenciada pelo sistema operacional, não importando qual o aplicativo a ser utilizado.

No estado da arte, referenciam-se conceitos de sistemas operacionais, sistemas de arquivos, proteção de arquivos e tipos de acesso.

Foram apresentadas as modificações realizadas na estrutura de abertura e criação de arquivos no sistema operacional Linux, destacando a inserção de um novo parâmetro, sendo este uma senha a ser armazenada em um campo reservado do inode do arquivo.

Na conclusão, destaca-se a importância deste trabalho visando a segurança de arquivos, sendo os mesmos protegidos por senha, controladas pelo sistema operacional, vinculadas ao proprietário do arquivo, porque somente o mesmo terá acesso a seus dados através de senha.

ABSTRACT

The objective of this work is to insert a password protection mechanism for files in the Linux operating system. Initially, it has been introduced operating systems concepts, file systems, file protection, and types of access. Then, it has been presented the modifications that were made in the creation and openning procedures of files in the Linux operating system, showing mainly the insertion of a password that can be kept in a reserved field of the file descriptor (known as the inode). Concluding, it has been stressed the importance of this work.

INTRODUÇÃO

A grande preocupação dos programadores de sistemas operacionais, além de otimização e bom aproveitamento dos recursos da máquina, adequando todo o sistema para atender às grandes inovações tecnológicas, é a segurança de que o usuário necessita para fazer uso desses recursos, sabendo que seu trabalho estará protegido de ataque ou de usuários sem permissão para obter tais informações.

Todos os sistemas operacionais trazem em seu conceito a segurança em relação a arquivos e diretórios. Uns constam com uma condição de segurança maior, outros não. Esta preocupação se torna maior quanto se trata de sistemas operacionais multiusuários, pois o controle de acesso deve ser redobrado. Com isso, é necessário que o conceito de segurança seja implementado logo no início da sessão de trabalho, solicitando ao usuário um login (nome para acesso) acompanhado de uma senha.

Nesse processo alguns bloqueios são instaurados através de um perfil do usuário onde são determinados os diretórios de trabalho e arquivos permitidos para acesso, podendo ser controlado até o horário em que poderá ficar conectado à estação principal de trabalho.

Em alguns casos, o sistema de proteção é inserido no arquivo através de uma rotina do próprio aplicativo que está sendo utilizado. Esta preocupação poderia ser minimizada transferindo esta tarefa para o sistema operacional, solicitando que o mesmo controle a inserção de senha na criação de arquivos.

1.1 OBJETIVO GERAL

A presente proposta de trabalho visa inserir no núcleo do sistema operacional Linux um mecanismo de proteção de arquivos através de senhas. Com esse mecanismo, o sistema operacional perguntaria ao proprietário do arquivo, quando da sua criação, se

o mesmo deseja inserir uma senha, a qual fará parte das permissões de acesso ao arquivo, e será independente do aplicativo usado.

A senha ficará vinculada ao inode do arquivo, onde o sistema operacional busca as informações dos mesmos.

1.2 JUSTIFICATIVA

Esta proposta de proteção implementada no núcleo de um sistema operacional é de grande importância prática para aumentar a segurança dos arquivos, pois somente o proprietário do arquivo poderá acessar suas informações. Com isso, o desenvolvimento de rotinas para implementar este processo de segurança em aplicativos será desnecessária.

Com esta implementação, o sistema de senha em arquivos receberá uma maior proteção, pois este controle será via sistema operacional e não por aplicativos. Estes aplicativos são vulneráveis, porque existe no mercado softwares que conseguem quebrar as senhas colocadas nos cabeçalhos dos mesmos.

Como o núcleo do sistema operacional possui uma proteção maior, esta proposta é de grande importância, levando-se em consideração a segurança das informações armazenadas em cada arquivo criado em um sistema operacional com esta característica.

Os estudos realizados para elaboração desta proposta, demonstram que não existem trabalhos desenvolvidos nesta área. As proteções de arquivos existentes no sistema operacional Linux decorrem apenas de senhas associadas ao login dos usuários.

2. ESTADO DA ARTE

2.1 SISTEMAS OPERACIONAIS

Nos últimos anos ocorreram mudanças substanciais na indústria da informática. Os computadores são hoje mais baratos e ao mesmo tempo mais poderosos. É possível encontrar no mercado desde computadores pequenos e baratos até computadores grandes e poderosos, mas o computador pessoal talvez seja o que mais atraiu a atenção do público. Os computadores pessoais podem ser usados para processar texto, confeccionar planilhas e ainda como entretenimento. No outro extremo estão os grandes computadores, que podem ser usados em processamento de imagens, previsão meteorológica, sistema de defesa e simulações de vôo.

Independente de seu tamanho, custo e capacidade, todos os computadores têm várias características em comum. Eles podem executar cálculos, tomar decisões e armazenar informações. As instruções necessárias para executar essas tarefas são ilegíveis para a maioria dos usuários.

O coração de todo computador é um conjunto de programas chamado de sistema operacional, que é o software que controla os sistemas de entrada/saída do computador como teclados e unidades de disco. Ele carrega e executa os outros programas. O sistema operacional também é um conjunto de mecanismos e dispositivos que ajudam e definem o compartilhamento e controlam os recursos do sistemas [GARFINKEL & SPAFFORD, 1996].

Um sistema operacional é um programa que age como um intermediário entre o usuário e o hardware do computador. A meta primária de um sistema operacional é tornar o sistema de computador conveniente para uso. Uma meta secundária é usar o hardware de computador de uma maneira eficiente. [SILBERSCHATZ & GALVIN, 1998]

Sistemas operacionais são principalmente os gerentes de recursos do computador, executam muitas funções, como implementar a interface de usuário e compartilhar hardware e dados entre os mesmos, prevenindo que usuários interfiram no processamento de aplicados de outros usuários, fazem a recuperação de erros, facilitam operações paralelas, organizam dados para acesso seguro e rápido. [DEITEL, 1990]

2.1.1 Tipos de sistemas operacionais

Como existem diversos tipos de computadores, consequentemente não é de se surpreender que existam muitos sistemas operacionais diferentes. Alguns, mais simples, dedicam todos os recursos do computador a um aplicativo de cada vez; outros permitem que o usuário execute múltiplos aplicativos simultaneamente.

Os sistemas operacionais podem ser monotarefa, multitarefa ou multiusuário.

O ambiente monotarefa é aquele onde o sistema executa um aplicativo de cada vez. Nesse caso todos os recursos do sistema ficam exclusivamente dedicados a uma única tarefa.

Consequentemente, se o programa aguarda por um evento, como a digitação de um dado, o processador ficará ocioso sem realizar qualquer tarefa. [MACHADO e MAIA, 1995]

Os sistemas multitarefa representam um avanço em relação aos sistemas monotarefa. Muitos ainda comportam apenas um usuário, que pode, entretanto, executar várias atividades ao mesmo tempo.

A multitarefa em um computador permite a execução simultânea de tarefas diferentes que anteriormente teriam de ser executadas seqüencialmente. O conjunto de tarefas não só é executado mais rapidamente como também computador fica livre para fazer outras coisas com o tempo economizado. [THOMAS & YATES, 1989]

Os sistemas multiusuário, são mais complexos do que os sistemas mono-usuários. O sistema operacional deve manter o controle dos dados de todos os usuários e impedir que eles interfiram uns com outros. A programação multitarefa tem a capacidade de rodar programas múltiplos ao mesmo tempo no computador. [MINASI et al., 1996]

2.2 SISTEMAS DE ARQUIVOS

Um sistema de arquivos é constituído pelos métodos e estruturas de dados que um sistema operacional utiliza para administrar arquivos em um disco ou partição.

Conceitualmente, arquivos são abstrações que fornecem meios de armazenar e recuperar informações em dispositivos periféricos. Arquivos podem ser armazenados pelo sistema operacional em diferentes dispositivos físicos, como fitas magnéticas, discos magnéticos e discos ópticos.

Um arquivo é constituído de informações logicamente relacionadas, podendo representar programas ou dados. Um programa contém instruções compreendidas pelo processador (arquivo executável), enquanto um arquivo de dados pode ser estruturado livremente, como em um texto, ou de forma mais rígida, como em um banco de dados. [MACHADO e MAIA, 2000]

Um arquivo é comumente identificado por meio de um nome, formado por uma seqüência de caracteres.

Um sistema de arquivos comporta arquivos e diretórios que são como objetos de sistemas. Estes objetos são representados através de sua descrição, podem ser uma coleção de arquivos, blocos, diretórios, e discriminação de arquivo, tudo em um disco lógico. É conveniente ter este objeto armazenado em local seguro. [CROWLEY, 1997]

O aspecto mais importante de um sistema de arquivo é como um arquivo aparece para ele, isto é, o que constitui um arquivo, como os arquivos são nomeados e protegidos, que operações são permitidas sobre o mesmo. [TANENBAUM & WOODHULL, 2000]

Para a maioria dos usuários, a interface do sistema de arquivo é o componente mais visível de um sistema operacional. Essa interface consiste de duas partes distintas: uma coleção de arquivos, e uma estrutura de diretório que organiza e provê informação sobre todos os arquivos no sistema. [SILBERSCHATZ & GALVIN, 1998]

2.2.1 Sistema Operacional DOS

O Sistema Operacional DOS organiza um disco flexível ou uma partição de disco rígido, mapeando-os em quatro áreas separadas, onde os dados serão armazenados na seguinte ordem: *área reservada, tabela de alocação de arquivos (FAT), diretório principal e a área de arquivos,* conforme figura n.º 2.1.

FIGURA n.º 2.1 – Organização de um disco flexível ou uma partição de disco rígido

Setor Lógico 0	Área reservada
	Tabela de Alocação de Arquivos (FAT)
	Diretório Principal
	Área dos arquivos (arquivos e subdiretórios)

A área reservada pode possuir um compartimento de um ou mais setores, sendo o primeiro setor sempre utilizado para partida (*boot*) do disco, seguida pela FAT que mapeia a utilização de todo o espaço do disco existente na área de arquivos, o espaço não-utilizado e o espaço que não pode ser utilizado devido a existência de defeitos no meio magnético do disco.

Esse sistema foi desenvolvido para o DOS 1.0 em 1981, quando o primeiro PC IBM foi lançado no mercado. Como o PC original não possuía discos rígidos, mas somente unidades de disco flexível de 5 ¼ polegadas para 160 KB, o sistema FAT foi elaborado para trabalhar de modo eficiente nesse ambiente limitado. À medida que os disquetes e discos rígidos cresceram em tamanho, as mudanças no sistema FAT não acompanharam adequadamente esse crescimento. Como é muito inconveniente para o sistema de arquivos FAT gerenciar individualmente as dezenas de milhares de setores em um disco muito grande, ele agrupa setores adjacentes em uma unidade de alocação chamada cluster, diminuindo assim o número de registros na FAT. Um cluster pode ter 2, 4, 8, 16, 32 ou 64 setores, dependendo do tamanho lógico do disco. O cluster é a menor quantidade de espaço em disco que pode ser alocada para um arquivo.

A FAT pode ser formatada com itens de 12 ou 16 bits. O formato de 12 bits será utilizado para discos flexíveis e para partições de disco rígido que contenham menos de 4078 clusters. Para determinar as condições de cada cluster a FAT utiliza alguns valores para sinalizar a condição de cada cluster no disco.

FIGURA n.º 2.2 - Os valores da FAT

Valores de 12 bits	Valores de 16 bits	Significado
000H	0000H	Cluster fora de uso
FF0H a FF6H	FFF0H a FFF6H	Cluster reservado
FF7H	FFF7H	Cluster defeituoso
FF8H a FFFH	FFF8H a FFFFH	Último Cluster de um arquivo

Fonte: NORTON, 1994.

O diretório principal será o item seguinte em qualquer disco de MS-DOS. Esse diretório funciona como se fosse uma tabela de conteúdo, identificando cada arquivo existente no disco com um item de diretório que contém várias peças de informação. Inclui a data e a hora em que cada arquivo foi criado ou atualizado, o tamanho do arquivo em bytes e o local no disco do primeiro cluster de dados que pertence ao arquivo.

A FAT demonstra como são alocados os agrupamentos de arquivos. O diretório mestre mostra a localização do inicio de cada arquivo, o tamanho, quando o arquivo foi criado, e qualquer atributo especial do arquivo. Os subdiretórios podem ser usados para organizar arquivos em disco. A maioria das aplicações não está familiarizada com este nível de detalhe. Elas chamam as rotinas de sistema operacional apropriadas, simplesmente para realizar tarefas desejadas, sem a preocupação de como as tarefas serão executadas e de como o dados serão organizados. [DEITEL, 1990]

2.2.2 Sistema Operacional Windows NT

O Windows NT suporta três tipos de sistemas de arquivo: FAT, HPFS e NTFS, conforme tabela a seguir:

TABELA 01 – Comparação dos sistemas de arquivo

CARACTERÍSTICA	FAT	HPFS	NTFS
Nomes de arquivos e diretórios	255 caracteres	254 caracteres	255 caracteres
Tamanho máximo de arquivos	4 Gbytes	4 Gbytes	16 Ebytes
Tamanho máximo da partição	4 Gbytes	2 Tbytes	16 Ebytes
Nível de segurança	Mínima	Média	Máxima
Estrutura de diretórios	Lista encadeada	Árvore-B	Árvore-B
Pode ser acessado	DOS, WinNT, Win95 e Win3.*	OS/2 e NT	WinNT

Fonte: MACHADO e MAIA (2000)

O sistema de arquivo NTFS foi projetado para o Windows NT com o intuito de oferecer alto grau de segurança e desempenho. Suas principais características são as seguintes [MACHADO e MAIA, 2000]:

- nomes de arquivos podem ter no máximo 255 caracteres, incluindo brancos;
- aceita letras maiúsculas e minúsculas, sem distingui-las;
- nomes de arquivos FAT podem suportar até 255 caracteres;
- partições NTFS dispensam o uso de ferramentas de recuperação de erros;
- implementa proteção de arquivos e diretórios, mas não implementa criptografia;
- menor partição recomendada é de 50 Mbytes;
- não pode ser implementado em disquetes;
- reduz a fragmentação no disco, na medida em que tenta sempre utilizar espaços contíguos de disco para a gravação de arquivos;
- suporta compressão de arquivos;
- tamanho máximo de arquivos de 64 Gbytes, mas podendo alcançar 16 Ebytes;
- tamanho máximo da partição de 2 Tbytes, mas podendo alcançar 16 Ebytes.

2.2.3 Sistema Operacional UNIX

O sistema de arquivo do UNIX controla o modo com que as informações dos arquivos são armazenados em diretórios e disco ou outra forma de armazenamento secundário. Controla se o usuário pode acessar o arquivo e como pode fazer isto. O sistema de arquivo é então uma das ferramentas básicas para obrigar a segurança de arquivos em um sistema. [GARFINKEL & SPAFFORD, 1996]

Muitos sistemas de arquivos do Unix tem uma estrutura similar, apesar dos detalhes variarem um pouco. Os conceitos básicos são superbloco, inode, bloco de dado, bloco de diretório e bloco de indireto.

O superbloco contém as informações sobre o sistema de arquivos como um todo, como por exemplo o seu tamanho (a informação exata depende do sistema de arquivos). Um inode contém informações sobre um determinado arquivo, exceto seu nome. O nome de um arquivo está armazenado no diretório, junto com o número do inode.

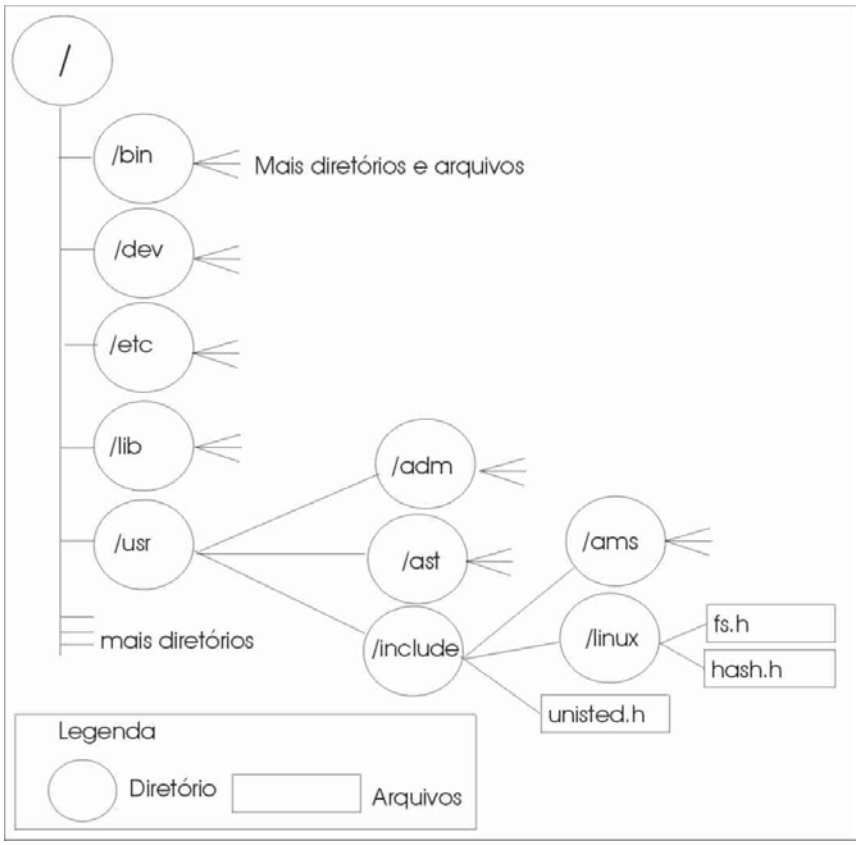
O Linux suporta diversos tipos de sistemas de arquivos, dentre os quais destacamos: minix, xia, ext, e ext2. Adicionalmente há o suporte a diversos outros sistemas de arquivos, para simplificar a troca de informações com outros sistemas operacionais. Estes sistemas de arquivos funcionam como se fossem nativos, exceto pela eventual perda de algumas facilidades presentes no Unix: msdos, umsdos, iso9660, nfs, hpfs e sysv.

A opção do sistema de arquivos a ser usado depende da situação. Caso a compatibilidade ou outras razões tornem um dos sistemas de arquivos não nativos necessário, então este deve ser utilizado. Caso a opção seja livre, provavelmente a decisão mais acertada seja usar o ext2, uma vez que ele traz diversas facilidades sem sofrer perda de desempenho.

Há ainda o sistema de arquivos proc, normalmente acessível através de diretório /proc, o qual não é um sistema de arquivo real, apesar de parecer um. O sistema de arquivos proc torna mais simples o acesso a determinadas estruturas do kernel, como por exemplo a lista de processos.

O sistema de arquivos do UNIX é baseado em uma estrutura de diretórios em árvore, sendo o diretório raiz (root) identificado por “/”.

FIGURA n.º 2.3 – Estrutura de diretórios



Fonte: AMARAL, 2002

No Unix não existe uma dependência entre a estrutura lógica dos diretórios e o local onde os arquivos estão fisicamente armazenados. Desta forma é possível adicionar novos discos ou partições ao sistema de arquivos sempre que necessário. Os atributos comuns são os mesmos, o que difere é que no Unix a quantidade de atributos é maior.

O sistema de arquivos do UNIX incorpora as seguintes características principais:

- Estrutura hierárquica – Os usuários podem agrupar informações congêneres e manipular eficientemente um grupo de arquivos como uma unidade.
- Ampliação de arquivo – Os arquivos crescem dinamicamente, conforme a necessidade, ficando apenas com o espaço exigido para armazenamento de seu conteúdo real. O usuário não é obrigado a decidir com antecedência o quanto um arquivo crescerá.

- Arquivos sem estrutura. O UNIX não impõe estrutura interna ao conteúdo de um arquivo. O usuário é livre para estruturar e interpretar o conteúdo de um arquivo de forma mais apropriada.
- Segurança. Os arquivos podem ser protegidos contra uso não autorizado de diferentes usuários do sistema UNIX.
- Arquivo e independência de dispositivo. O UNIX dá tratamento idêntico para arquivos e para entrada/saída. Os mesmos procedimentos e programas usados para processar informações armazenadas de arquivo podem ser usados para ler dados de um terminal, imprimi-los e passá-los para um outro programa. [http://br.tldp.org/documentos/livros/html/gas/node44.html, 2000]

O sistema UNIX de arquivamento permite o controle flexível sobre o acesso a programas e dados. O acesso para o proprietário, o acesso para grupos definidos de usuários e o acesso para todos os usuários do sistema podem ser controlados independentemente para cada arquivo criado. [THOMAS & YATES, 1989]

No Unix, diretórios são implementados como arquivos com um formato especial e, portanto, a representação de um arquivo é o conceito básico.

Cada arquivo possui seus atributos em uma estrutura chamada INODE.

O inode contém a identificação do usuário e do grupo do arquivo, as datas da última modificação e do último acesso, um contador do número de vínculos (entradas de diretório) do arquivo e o tipo do arquivo (arquivo regular, diretório, vínculo simbólico, etc).

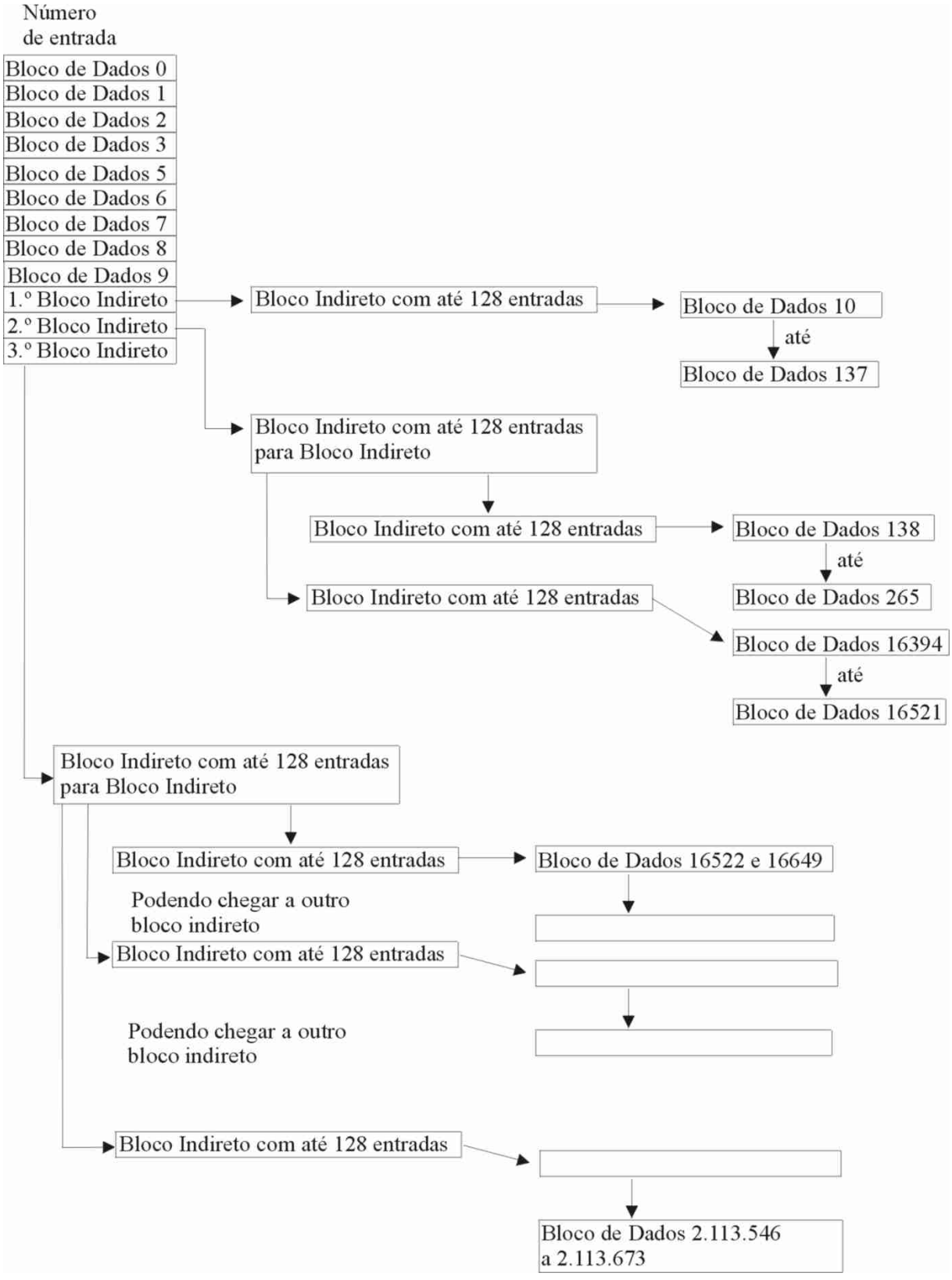
Um inode tem até treze ponteiros. Os primeiros dez contêm as posições dos blocos de dados do arquivo. Se houver menos de dez blocos, o UNIX usará tantos ponteiros quantos forem necessários. Assim, os primeiros dez ponteiros podem localizar até 5K (10 blocos * 0,5K por bloco) de dados.

O 11º ponteiro contém a posição de um primeiro *bloco indireto.* O bloco indireto contém até 128 entradas e cada uma pode conter a posição de outro bloco de dados. Assim, o bloco indireto pode especificar as posições de até 128 blocos de dados. Esse bloco de dados adicional permite que os arquivos tenham até 5KB + 64KB = 69 Kbytes.

O 12º ponteiro contém a posição de um *segundo bloco indireto.* Ele tem outras 128 entradas, e cada uma delas pode conter a posição de outro bloco indireto. Assim, pode haver mais 128 blocos indiretos. Cada um, por sua vez, pode conter as posições de até 128 blocos de dados. Assim, o segundo bloco indireto permite ao UNIX acessar até mais 128 * 128 = 16.384 blocos, ou mais 8.l92 Kbytes de dados. Um arquivo agora pode chegar até 69KB + 8192KB = 8261KB, ou 8,261 megabytes (MB).

O 13º ponteiro contém a posição de um terceiro bloco indireto. Ele tem outras 128 entradas. Cada uma delas, conforme descrevemos, contém as posições de até mais 128 blocos indiretos, e podem localizar até mais 16.384 blocos de dados. Como o terceiro bloco indireto pode localizar até 128 blocos duplamente indiretos, ele pode proporcionar acesso até 128 * 16.384 = 2.097.152 blocos de dados. Isso dá um adicional de 1.048.576 bytes, ou mais de 1 gigabyte (GB = 1 bilhão de bytes) de dados. Assim os arquivos do UNIX podem ter até 1,056837 GB.

FIGURA 2.4 – Acessos do UNIX aos blocos de dados



Fonte: AMARAL, 2002

2.3 SISTEMA DE PROTEÇÃO DE ARQUIVO

Todo aplicativo de computador precisa armazenar e recuperar informações. Esta atividade é essencial para qualquer tipo de aplicação. Um processo deve ser capaz de ler e gravar dados em dispositivos como fitas e discos de forma permanente. É mediante a implementação de arquivos que o sistema operacional estrutura e organiza estas informações.

O sistema de arquivos, é responsável pela gerência, através do sistema operacional, que para o usuário é transparente.

Um arquivo é constituído de informações logicamente relacionadas, podendo representar programas e dados. Na realidade um arquivo pode ser entendido como um conjunto de registros definidos pelo sistema de arquivos.

Em alguns sistemas operacionais, a identificação de um arquivo é composta por duas partes separadas com um ponto. A parte após o ponto é denominada extensão do arquivo e tem como finalidade identificar seu conteúdo.

FIGURA n.º 2.5 – Algumas típicas extensões de arquivo.

Extensão	Significado
.bak	Arquivo de backup
.c	Programa fonte em C
.gif	Imagem do formato GIF (Graphical Interchange Format)
.cob	Arquivo fonte em COBOL
.zip	Arquivo compactado

Considerando que os meios de armazenamento são compartilhados entre diversos usuários, é de fundamental importância que mecanismos de proteção sejam implementados para garantir a proteção individual de arquivos e diretórios. Qualquer sistema de arquivos deve possuir mecanismos próprios para proteger o acesso às informações em discos e fitas, além de possibilitar o compartilhamento de arquivos entre usuários, quando desejado.

Cada arquivo possui informações de controle denominadas atributos. Dependendo do sistema de arquivos, os atributos variam, porém alguns, como tamanho do arquivo, proteção, identificação do criador e data/hora de criação, estão presentes em quase todos os sistemas.

Alguns atributos especificados na criação do arquivo não podem ser modificados em função de sua própria natureza, como organização e data/hora de criação. Outros são alterados pelo próprio sistema operacional, como tamanho e data/hora do último backup realizado. Existem ainda atributos que podem ser modificados pelo próprio usuário, como proteção do arquivo, tamanho máximo e senha de acesso.

A Tabela 02 lista possíveis atributos de arquivos.

TABELA 02 – Lista de possíveis atributos de arquivos

Campo	Significado
Proteção	Quem pode acessar o arquivo e de que maneira
Senha	Senha necessária para acessar o arquivo
Criador	Id da pessoa que criou o arquivo
Proprietário	Proprietário atual
Sinalizador de somente-leitura	0 para leitura/gravação; 1 para somente leitura
Sinalizador de oculto	0 para normal; 1 para não exibir em listagens
Sinalizador de sistema	0 para arquivo normal; 1 para arquivo de sistema
Sinalizador de arquivo	0 para salvo em backup; 1 para ser salvo em backup
Sinalizador de ASCII/binário	0 para arquivo ASCII; 1 para arquivo binário
Sinalizador de acesso aleatório	0 p/ acesso seqüencial somente; 1 p/ acesso aleatório
Sinalizador acesso temporário	0 p/ arquivo normal; 1 p/ excluir arquivo na saída do processo
Sinalizador de bloqueio	0 para destravado; não-zero para bloqueado
Comprimento do registro	Número de bytes em um registro
Posição da chave	Deslocamento da chave dentro de cada registro
Tempo de criação	Data e hora em que o arquivo foi criado
Tempo do último acesso	Data e hora em que o arquivo foi acessado pela última vez
Tempo da última alteração	Data e hora em que o arquivo foi alterado pela última vez
Tamanho atual	Número de bytes no arquivo
Tamanho máximo	Número de bytes até o qual o arquivo pode crescer

Fonte: TANENBAUM & WOODHULL (2000)

2.3.1 Sistema Operacional DOS

Os arquivos podem ser acessados por uma estrutura chamada de Controle de Bloco de Arquivos (FCB) que faz a chamada ou manuseio do arquivo (Versão 2.0 e posteriores); são chamadas projetadas para manuseio de arquivo para trabalhar com um sistema de arquivo hierárquico. As chamadas de arquivos são preferencialmente, providas pelo FCB em compatibilidade com as versões prévias do MS-DOS.

No FCB funcionam pedidos como segue:

TABELA 03 – Funções dos FCBs

Decimal	Descrição
15	Abrir arquivo
16	Fechar arquivo
17	Localizar o primeiro item de diretório que satisfaça à condição
18	Localizar o próximo item de diretório que satisfaça à condição
19	Eliminar arquivo
20	Leitura seqüencial
21	Gravação seqüencial
22	Criar arquivo
23	Renomear arquivo
33	Leitura de registro aleatório
34	Gravação de registro aleatório
35	Obtém o tamanho do arquivo
36	Define o campo de registro aleatório do FCB
39	Leitura de registros aleatórios
40	Gravação de registros aleatórios
41	Dividir um nome de arquivo

Fonte: NORTON, 1994

A estrutura de dados do FCB normal ocupa um espaço da memória de aplicação e contem 37 bytes de dados para controlar os processos de entrada/saída de arquivos, um FCB estendido de 44 bytes será utilizado por algumas funções do MS-DOS: 7 bytes serão acrescentados ao início de uma estrutura de dados normal.

Segue tabela com a sua descrição:

TABELA 04 – Estrutura de um bloco estendido de controle de arquivo

Decimal	Largura do Campo	Descrição
0	1	Flag de FCB estendido (sempre FFH)
1	5	(Reservado)
6	1	Atributo
7	1	Identificação (0=default drive, l=drive UM, 2=drive B, e assim por diante)
8	8	Nome do arquivo
16	3	Extensão do arquivo
19	2	Número atual de bloco
21	2	Tamanho do registro em bytes
23	4	Tamanho do arquivo em bytes
27	2	Data
29	2	Horário
31	8	(Reservado)
39	1	Número de registro atual
40	4	Número de registro aleatório

Fonte: NORTON, 1994

2.3.2 Sistema operacional Unix

Cada arquivo tem necessariamente um nome e um conjunto de dados. Além disso, o sistema operacional associa a cada arquivo algumas outras informações que chamaremos de atributos de arquivos.

O sistema de arquivos do Linux permite restringir o acesso aos arquivos e diretórios permitindo que somente determinados usuários possam acessá-los. A cada arquivo e diretório é associado um conjunto de permissões.

Essas permissões determinam quais usuários podem ler, escrever ou alterar um arquivo, e no caso de arquivos executáveis, quais usuários podem executá-lo. (Se um usuário tem permissão de execução de um diretório, significa que ele pode realizar buscas dentro daquele diretório, e não executá-lo como se fosse programa).

O domínio de um processo é definido por seu UID e seu GID. Dada qualquer combinação (uid, gid), é possível fazer uma lista completa de todos os objetos (arquivos, incluindo dispositivos de E/S representados por arquivos especiais, etc.) que podem ser acessados para leitura, para gravação ou para execução. [TANENBAUM & WOODHULL, 2000]

O sistema UNIX fornece várias proteções para a segurança do sistema, incluindo senha para proteger o acesso ao sistema, controle de acesso aos arquivos individuais, codificação dos arquivos de dados e recursos administrativos para uma análise das alterações realizadas pelo usuário sob um arquivo. [THOMAS & YATES, 1989]

Como implementação de proteção de arquivos, o mecanismo de proteção utiliza-se de domínios de proteção, que é um conjunto de pares (objeto, direitos), que especifica um objeto, e um subconjunto das operações que podem ser realizadas sobre um arquivo. Necessidade de um direito que neste contexto significa a permissão para efetuar operações como READ e WRITE e EXECUTE sobre o arquivo.

A cada instante, todo processo roda em determinado domínio de proteção. Existe um conjunto de objetos que ele pode acessar, e sobre cada um desses objetos, tem-se um conjunto de direitos. Cada processo pode mudar de domínio no decorrer de sua execução. Estas regras de troca de domínio são determinadas pelo sistema operacional.

O domínio é um processo que definido por parâmetros UID e GID, dessa combinação. Pode-se elaborar uma lista completa de todos os objetos que podem ser acessados e se podem ser acessados para leitura, escrita e execução.

Cada processo no UNIX é dividido em duas partes, uma pertencente ao usuário e outra ao kernel.

2.4 ESTRUTURA DE ARQUIVOS

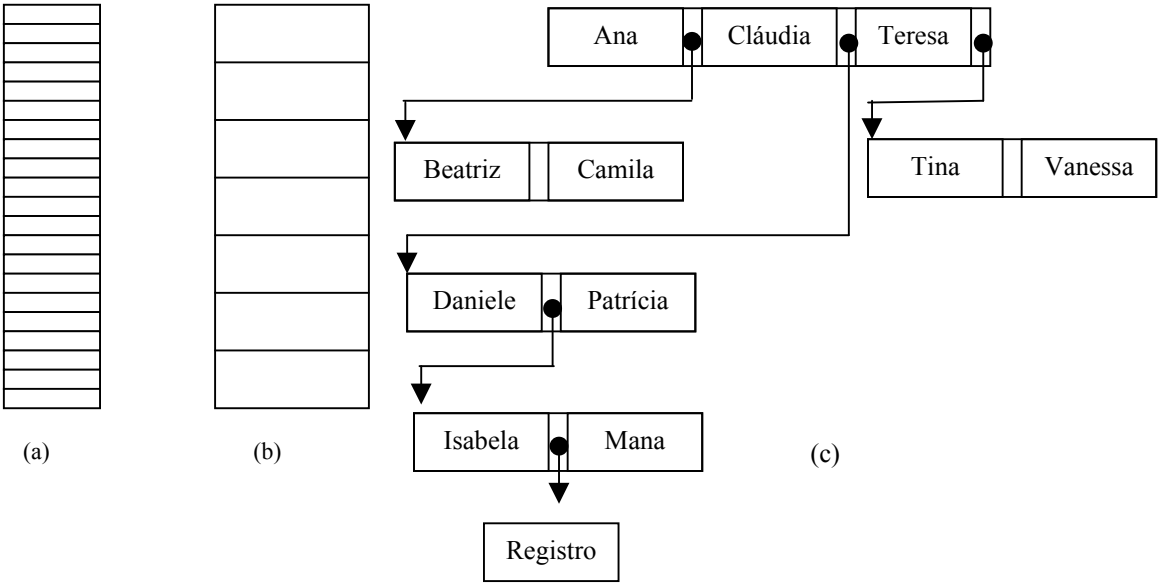
A organização de arquivos consiste no modo como os seus dados estão internamente armazenados. A estrutura dos dados pode variar em função do tipo de informação contida no arquivo.

No momento da criação de um arquivo, é possível definir que organização será adotada. Esta organização pode ser uma estrutura suportada pelo sistema operacional ou definida pela própria aplicação.

A forma mais simples de organização de arquivos é através de uma seqüência não-estruturada de bytes (figura 2.6a). Nesse tipo de organização, o sistema de arquivos não impõe nenhuma estrutura lógica para os dados.

Alguns sistemas operacionais estabelecem diferentes organizações de arquivos. Neste caso, cada arquivo criado deve seguir um modelo suportado pelo sistema de arquivos.

FIGURA 2.6 – Organização de Arquivos



Fonte: TANENBAUM, 1999

No modelo de estrutura de arquivo de seqüência de registro, o mesmo mantém o tamanho do registro fixo (figura 2.6b).

Outra estrutura de organização é composta de uma árvore de registros, contendo um campo chave numa posição fixa do registro. A árvore é ordenada pelo campo chave, de modo a permitir uma busca rápida de determinada chave (figura 2.6c).

2.5 MÉTODOS DE ACESSO

Em função de como o arquivo está organizado, o sistema de arquivos pode recuperar registros de diferentes maneiras. Inicialmente, os primeiros sistemas operacionais só armazenavam arquivos em fitas magnéticas. Com isso, o acesso era restrito à leitura dos registros. Era realizada na ordem em que eram gravados, sendo a gravação de novos registros possível apenas no final do arquivo. Este tipo de acesso,

chamado de seqüencial, era próprio da fita magnética, que como meio de armazenamento possuía tal limitação.

Com o advento dos discos magnéticos, foram introduzidos métodos de acesso mais eficientes. O primeiro a surgir foi o acesso direto, que permite a leitura/gravação de um registro diretamente na sua posição.

Este acesso foi denominado acesso aleatório, estes arquivos são fundamentais para a implementação de uma série de aplicações. Existem duas maneiras de se especificar onde uma leitura deve começar. Na primeira delas, cada operação de READ deve especificar a posição do arquivo onde se deve começar a ler. Na segunda, a operação especial de SEEK deve ser usada para encontrar a posição desejada, depois disso, o arquivo pode ser lido seqüencialmente a partir da posição.

Outra maneira é usar uma *função hash* para calcular uma posição dentro de um arquivo com base no valor de índice de campo.

Para acesso a informações em um arquivo indexado por árvore, os registros podem ser acessados através de uma estrutura hierárquica. Cada registro pode ser rapidamente acessado de forma seqüencialmente individual, por ordem de campo de chave. Esta estrutura baseada em árvore define cada setor do disco como um nó dessa árvore. O sistema armazena todos os registros em setores que correspondem a nós-folhas, estes nós contêm somente registros estando sempre no mesmo nível da árvore, formando os nós de dados.

O sistema armazena registros em um nó de dados de acordo com o valor de seus campos de chave de índice. Para que os valores fiquem em ordem crescente os índices são armazenados da esquerda para direita.

Para esta ordenação chamamos esta árvore e árvore-B de ordem k, pois seus valores ficam entre k/2 e k. Isto simplifica os algoritmos de inserção e exclusão.

2.6 OPERAÇÕES COM ARQUIVOS

O sistema de arquivo oferece um conjunto de chamadas de sistema que permite às aplicações realizar operações de entrada/saída, como tradução de nomes em endereços, leitura e gravação de dados e criação/eliminação de arquivos.

Sistemas diferentes oferecem operações diferentes para permitir armazenamento e recuperação. As operações mais comuns com arquivos são:

FIGURA n.º 2.7 – Operações de entrada/saída

Comando	Descrição
Create	Permite a criação de arquivo sem dados. O propósito desta chamada é anunciar a criação de um arquivo e especificar alguns de seus atributos
Delete	Eliminação de um arquivo para liberar espaço em disco. Existem sistemas que deletam automaticamente qualquer arquivo não usado por mais de n dias.
Open	Permite que o sistema busque os atributos e a lista dos endereços em disco correspondente, carregando estas informações na memória principal, informações necessárias para sua abertura.
Close	Fechamento de um arquivo, quando não houve acesso ao mesmo, seus atributos e endereços no disco não precisam mais ser mantidos na memória principal, de forma que o arquivo pode ser fechado.
Read	Dados são lidos do arquivo. Usualmente, os bytes lidos vêm da posição corrente de leitura. O processo que faz a chamada deve indicar a quantidade de informação a ser lida e providenciar um buffer para possibilitar a leitura.
Write	Os dados são escritos no arquivo, geralmente a partir da posição corrente. Se tal posição for a de final de arquivo, o tamanho do mesmo cresce. Se a posição do momento da escrita estiver no meio do arquivo, os dados existentes nesta posição estarão perdidos para sempre, pois a operação de escrita escreve os novos dados em cima dos antigos.
Append	Esta chamada é uma forma restrita do WRITE. Ela só pode adicionar dados no final do arquivo. Sistemas só fornecem uma quantidade mínima de chamadas, seguramente não têm o APPEND entre suas chamadas.
Seek	Para arquivos de acesso randômico. Trata-se de uma chamada que especifica o ponto a partir do qual os dados devem ser acessados. Uma forma comum de implementar esta chamada é reposicionar o ponteiro da posição corrente para um lugar especificado na própria chamada.
Get Attributes	Os processos muitas vezes precisam ler os atributos dos arquivos para realizarem seu trabalho.
Set Attributes	Alguns atributos podem ser setados pelo próprio usuário e podem ser modificados após a criação do arquivo.
Rename	Alteração do nome de um arquivo.

Fonte: TANENBAUM (1999)

2.7 ATRIBUTOS

2.7.1 Proteção de Acesso

Cada arquivo no Unix apresenta três níveis de proteção definidos por três categorias de usuários. Todo arquivo ou diretório tem um dono (user) e pertence a um grupo (group). Qualquer usuário do sistema que não seja o dono do arquivo e não pertença ao grupo enquadra-se na categoria outros (others). Para cada categoria de usuário, três tipos de acesso podem ser concedidos: leitura (read), gravação (write) e execução (execute).

2.7.2. Permissões de Arquivos

Além dos atributos básicos que um arquivo pode ter (read, write e execute), existem dois outros atributos de extrema relevância para a segurança, que são:

a) SUID: Este atributo, quando definido em um arquivo, indica que este, caso seja um executável ou um script, quando do momento de sua execução, terá o seu userid atribuído de acordo com o userid de quem é o proprietário do arquivo, ao invés de quem o executa. Este tipo de privilégio é especialmente perigoso, pois um arquivo cujo dono seja o root, com SUID setado e que possa ser executado por qualquer usuário, independente de quem o execute, terá os privilégios de um processo do próprio root.

b) SGID: Este atributo, quando definido em um arquivo, similarmente ao SUID, fará com que o processo herde os privilégios do grupo do proprietário do arquivo. Apesar de muitas distribuições do Linux, por padrão, ajustarem as permissões dos arquivos especiais do sistema, é sempre aconselhável tomar certos cuidados:

Em todos os arquivos de log do sistema, como, por exemplo, o arquivo /var/log/messages e /var/log/secure, ajustar suas permissões para 600 (-rw------), o que significa que apenas root poderá ler e gravar nestes arquivos. Isto porque nestes

arquivos podem aparecer informações sobre possíveis erros no sistema, que podem vir a ser explorados pelo atacante.

Verificar também se os arquivos de configuração do sistema (geralmente situados no diretório /etc) possuem o atributo de gravação definido apenas para root (644).

Existe um atributo pouco conhecido, que na verdade diz respeito apenas ao sistema de arquivos do Linux (ext2), que permite que um arquivo nunca possa ser modificado/apagado. Este atributo pode ser definido e removido apenas pelo root e pode ser uma forma interessante de reforçar a segurança de arquivos críticos. Isto pode ser feito através do comando "chattr +i file". Como root, verificar a PATH e certificar-se de que realmente esteja executando o programa que deseja, no local em que este deveria estar. Há, na maioria das listas de bugs de segurança, relatos sobre incidentes onde o root inadvertidamente executa programas "maliciosos" do usuário, pensando serem programas do sistema. Procurar por arquivos que possuam os SUID ou SGID definidos; eles podem ser uma potencial fonte de problemas. Para fazer isto, executa-se o comando "find / -type f \(-perm -04000 -o -perm -02000 \)", que irá listar todos os arquivos com algum destes privilégios definidos.

Arquivos e diretórios que podem ser gravados por qualquer usuário (atributo w setado para outros) são um considerável ponto de vulnerabilidade. Executar um comando "find / -perm -2 -print" e verificar se esta permissão realmente é necessária para estes arquivos. Antes de mudar a permissão de qualquer arquivo do sistema, primeiro tenha certeza do que esteja fazendo. Nunca mude as permissões de um arquivo para tornar seu trabalho mais fácil. Sempre verifique porque aquele arquivo tem aquelas permissões, antes de mudá-las.

Considerando que os meios de armazenamento são compartilhados entre diversos usuários, é de fundamental importância que mecanismos de proteção sejam implementados para garantir a proteção individual de arquivos e diretórios. Qualquer sistema de arquivos deve possuir mecanismos próprios para proteger o acesso às informações gravadas em discos e fitas, além de possibilitar o compartilhamento de arquivos entre usuários, quando desejado.

Existem diferentes mecanismos e níveis de proteção, cada qual com suas vantagens e desvantagens, sendo que, para cada tipo de sistema, um modelo é mais adequado do que outro.

2.7.2.1 Códigos de Proteção

Todo arquivo determina quais usuários têm acesso a ele e com que finalidade. Cada categoria de usuários possui um conjunto distinto de permissões de acesso ao arquivo. Cada conjunto de permissões de acesso significa presença ou ausência de permissões para:
Leitura (**r**)
Escrita (**w**)
Execução (**x**)

2.7.2.2 Verificando Permissões

Cada usuário do sistema possui três conjuntos (**rwx**) de permissão para cada arquivo. O sistema de permissões dá ao usuário mais segurança, pois permite que ele tenha um maior controle ao acesso de seus arquivos e diretórios. Isto dá mais segurança não só ao usuário, mas a todo o sistema. As permissões são mostradas quando se listam os arquivos usando o formato longo do comando ls (**ls -l**):
[root@mail /root]# ls -l conteudo.
***-rwxrw-r-- 1 root bin 1619 Aug 3 14:02 conteudo.bk**

Note que para o arquivo acima, o dono possui permissão de leitura(**r**), escrita(**w**) e execução(**x**). O grupo possui permissão de leitura(**r**) e escrita(**w**). Outro usuário que não o dono e não pertencente ao grupo do dono possui permissão somente de leitura(**r**).

2.7.2.3 Definindo Permissões de acesso em Arquivos e ou Diretórios

Para se definir as permissões de acesso a arquivos no Unix, podemos utilizar o modo simbólico ou absoluto

No modo simbólico utilizamos uma lista de expressões em que devemos informar um identificador sendo:

u – permissão para o dono do arquivo

g – permissão para o grupo do arquivo

o – permissão para outros

a – permissões para todos: proprietário, grupo e outros.

É necessário, além do identificador, um operando que determina a adição (+), remoção (-) ou a definição de uma nova permissão, cancelando as existentes (=). Estes operandos serão informados após indicação do identificador e antes da descrição da permissão.

As permissões utilizadas nos arquivos serão listadas no final da instrução sendo:

- nenhuma permissão

r – permissão de leitura

w – permissão de escrita

x – permissão de execução

s – bit setuid se for atribuído a u, setgid se for atribuído a g.

T – bit sticky

Para o modo absoluto, as permissões representam, por um número Octal de quatro dígitos, a forma com que as permissões são apresentadas:

e – atributo especial

u – permissão para o dono do arquivo

g – permissão para o grupo do arquivo

o – permissão para outros.

Na forma octal, os identificadores de usuário (u), grupo (g) e outros (o) serão representados por um número conforme segue:

0 – nenhuma permissão

1 – permissão de execução

2 – permissão de escrita

3 - permissão de escrita e execução

4 - permissão de leitura

5 - permissão de leitura e execução

6 - permissão de leitura e escrita

7 - permissão de leitura, escrita e execução

Para o dígito especial (e) mencionado no modo absoluto, os valores em octal representam:

0 – nenhum atributo especial ligado

1 – bit sticky ligado

2 – bit setgid ligado

3 – bits sticky e setguid ligados

4 – bit setuid ligado

5 – bit sticky e setuid ligados

6 – bit setuid e setgid ligados

7 – sticky, setuid e setgid ligados

Para este atributo especial informamos os significados de:

Sticky - Um arquivo criado sob um diretório com o bit **sticky** ligado pode ser apagado apenas por seu proprietário. Um programa com o bit **sticky** ligado terá seu texto mantido na área de **swap** do sistema.

Setuid - O arquivo é executado como se fosse invocado pelo proprietário; não faz sentido para diretórios.

Setgid - O arquivo é executado sob seu grupo, mesmo que o usuário invocador não faça parte dele; todo arquivo criado em um diretório **setgid** é criado com o mesmo grupo do diretório.

Analisando as permissões elencadas anteriormente no modo simbólico e absoluto, poderemos definir as permissões dos arquivos utilizando o comando chmod.

chmod u+x arquivo -> adiciona a permissão de execução para o proprietário do arquivo.
chmod ug+rw arquivo -> adiciona a permissão de leitura e escrita para o proprietário e grupo.
chmod u+wx,g-w,o=r arquivo -> adiciona a permissão de escrita e execução para o proprietário, retira a permissão de escrita para o grupo e atribui apenas a permissão de leitura para os outros usuários.

As permissões atribuídas a arquivos no modo octal utilizam a descrição de um digito em substituição aos caracteres do modo simbólico.

Para definir as permissões de Leitura, Escrita e Execução para o Proprietário, Leitura e Escrita para o Grupo, e somente Leitura para os demais (Outros) usuários, utilizando o comando chmod, temos:

 # chmod 764 arquivo

2.7.2.4 Alterando o Grupo dono dos arquivos/diretórios

O comando "**chgrp**" muda o Grupo dono dos arquivos e ou diretórios dados como argumento. O parâmetro **group** pode ser tanto um número (gid - identificador de grupo), como um nome de grupo encontrado no arquivo de grupos do sistema "/etc/group".

Para executar este comando, o usuário deve ser membro do grupo especificado e dono do arquivo (ou o superusuário).

Para comparar um arquivo qualquer antes e depois da execução do "chgrp":

Antes:

[root@mail /root]# ls -l arquivo

-rw-rw-rw- 1 mickey **disney** 41 Sep 24 17:47 arquivo

[root@mail /root]# chgrp root arquivo

Depois:

-rw-rw-rw- 1 mickey **root** 41 Sep 24 17:47 arquivo

2.7.2.5 Alterando o Dono dos arquivos/diretórios

O comando "**chown**" muda o dono dos arquivos e ou diretórios para um novo dono, que pode ser um nome de acesso ou a identificação de usuário (número associado ao nome do usuário).

Para comparar um arquivo qualquer antes e depois da execução do "chown":

Antes:

[root@mail /root]# ls -l arquivo

-rw-rw-rw- 1 **mickey** root 41 Sep 24 17:47 arquivo

[root@mail /root]# chown root arquivo

Depois:

-rw-rw-rw- 1 **root** root 41 Sep 24 17:47 arquivo

2.8 DIRETÓRIOS

A estrutura de diretório é o modo como o sistema organiza logicamente os diversos arquivos contidos em um disco. Quando um arquivo é aberto, o sistema operacional procura a sua entrada na estrutura de diretórios, armazenando as informações sobre atributos e localização do arquivo em uma tabela mantida na memória principal. Esta tabela contém todos os arquivos abertos, sendo fundamental para aumentar o desempenho das operações com arquivos.

As operações realizadas nos diretórios são as mesmas realizadas nos arquivos, com os mesmos atributos.

O sistema operacional mantém essas informações em diretórios. Se relacionados a informações de usuários, chamam-se diretórios de contas. Se se referem aos arquivos do usuários, chamam-se diretório de arquivos.

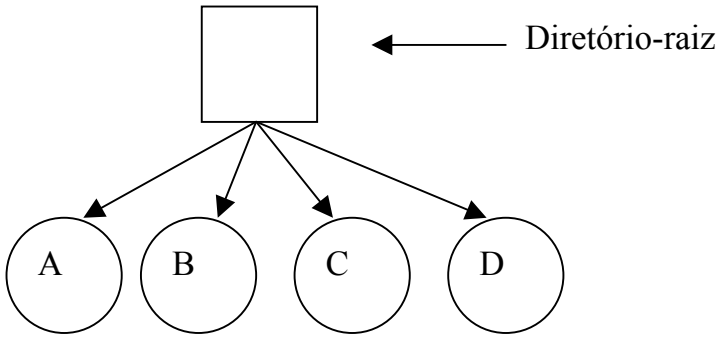
Os computadores têm diferentes tipos de diretórios, contendo diferentes informações, a estrutura dos diretórios simples, conhecida como diretório linear. Esta estrutura fornece uma maneira muito simples de gerenciar informações sobre contas e

arquivos. Para melhorar a organização das informações é necessária uma abordagem top-down. Este projeto usa uma organização chamada hierárquica.

2.8.1 Diretório Linear

Esta implementação é a mais simples na organização de uma estrutura de diretório, chamada de nível único, pois existe apenas um único diretório contendo todos os arquivos do disco. Este modelo é bastante limitado, já que não permite que usuários criem arquivos com o mesmo nome, o que ocasionaria um conflito no acesso de arquivos.

FIGURA n.º 2.8 – Estrutura de diretórios de nível único



Fonte: TANENBAUM, (1999)

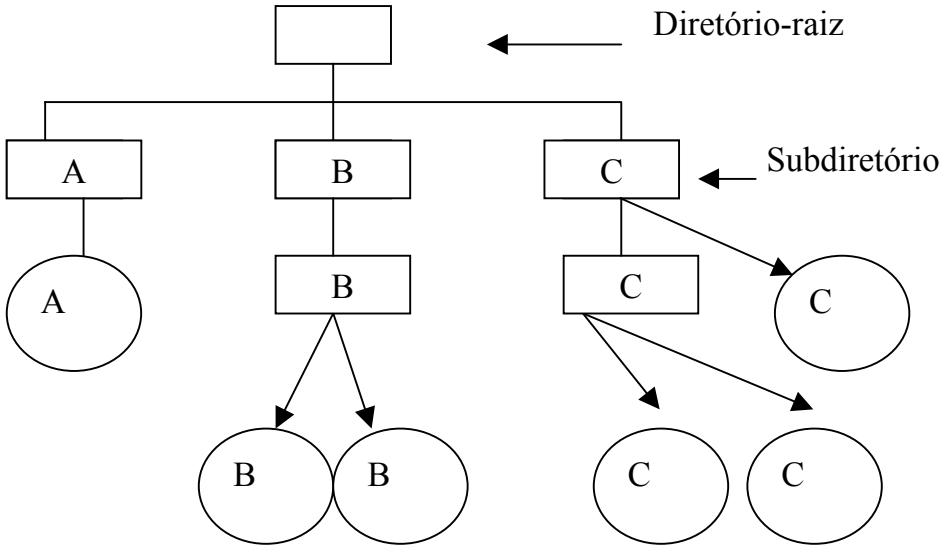
Um registro de conta contém tipicamente as seguintes informações: número da conta, nome do usuário, senhas, nível de prioridade, limites - como limites de CPU ou limites de memória e posição de um diretório de arquivo.

2.8.2 Diretório Hierárquico

No diretório hierárquico, existem diferentes níveis, sendo o topo o diretório raiz. Um usuário que esteja no diretório-raiz em geral vê somente as entradas do próximo ou segundo nível de diretórios. Este segundo nível é chamado de subdiretório.

Um subdiretório, por sua vez, pode conter entradas de arquivos e de subdiretórios subordinados.

FIGURA n.º 2.9 - Estrutura de diretórios de vários níveis



Fonte: TANENBAUM, (1999)

Uma vantagem é que o usuário que está em um subdiretório vê somente o que está nele. Definindo e nomeando subdiretórios adequadamente, o usuário pode colocar várias posições na hierarquia.

2.9 LISTA DE CONTROLE DE ACESSO

A lista de controle de acesso consiste em uma lista associada a cada arquivo, onde são especificados quais os usuários e os tipos de acesso permitidos. Neste caso, quando um usuário tenta acessar um arquivo, o sistema operacional verifica se a lista de controle autoriza a operação desejada.

O tamanho dessa estrutura de dados pode ser bastante extenso se considerarmos que um arquivo pode ter seu acesso compartilhado por diversos usuários. Além deste fato, existe um *overhead* adicional, se comparado com o mecanismo de proteção por grupo de usuários, devido à pesquisa que o sistema deverá realizar na lista sempre que um acesso for solicitado.

Em determinados sistemas de arquivos é possível encontrar tanto o mecanismo de proteção por grupos de usuários quanto o de lista de controle de acesso, oferecendo, desta forma, uma maior flexibilidade ao mecanismo de proteção de arquivos e diretórios.

Esta lista pode ser implementada por uma matriz, que conterá todas as informações dos arquivos, uma matriz de proteção conforme figura abaixo:

FIGURA nº 2.10 – Matriz de proteção

	Objeto							
	Arquivo1	Arquivo2	Arquivo3	Arquivo4	Arquivo5	Arquivo6	Impressora1	Plotter 2
1	Leitura	Leitura Escrita						
2			Leitura	Leitura Escrita Execução	Leitura Escrita		Escrita	
3						Leitura Escrita Execução	Escrita	Escrita

Fonte: TANENBAUM (1999)

Esta matriz pode ser representada com freqüência por dois métodos um por linha outro por coluna, que armazenam somente as posições ocupadas.

A primeira técnica consiste em associar a cada objeto uma lista (ordenada) contendo todos os domínios que podem acessar o objeto, juntamente com a forma de acesso permitida.

FIGURA nº 2.11 – Matriz de proteção com domínios como objetos

	Objeto										
Dom	Arq1	Arq2	Arq3	Arq4	Arq5	Arq6	Imp1	Plotter 2	Dom1	Dom2	Dom3
1	Lei	Lei Esc								Ent	
2			Lei	Lei Esc Exe	Lei Esc		Esc				
3						Lei Esc Exe	Esc	Esc			

Legenda

Dom – Domínio	Arq2 – Arquivo2	Imp1 – Impressora1
Dom1 – Domínio1	Arq3 – Arquivo3	Lei – Leitura
Dom2 – Domínio2	Arq4 – Arquivo4	Esc – Escrita
Dom3 – Domínio3	Arq5 – Arquivo5	Exe – Execução
Arq1 – Arquivo1	Arq6 – Arquivo6	Ent - Entrada

Fonte: TANENBAUM (1999)

Para visualizarmos esta lista de acesso, necessitamos assumir a existência de usuários com direitos e restrições sobre os arquivos e objetos.

Cada usuário recebe suas permissões para ler, escrever e executar o arquivo, onde a instrução a ser executada depende do tipo de permissão atribuída, podendo o proprietário de um objeto mudar a qualquer momento a lista de controle de acesso, tornando fácil proibir acessos antes permitidos.

2.10 MECANISMOS DE SENHAS EM PROGRAMAS APLICATIVOS

A preocupação com a segurança das informações gravadas em arquivos, tanto texto como banco de dados, aumenta a cada dia. Tal preocupação é visível em empresas ou no trabalho de profissionais que necessitam de segurança para proteger seus dados, antes que seu trabalho possa ser apresentado.

Sistemas de proteção por senhas são comumente utilizados em aplicativos. Neste caso, a senha é criptografada e gravada no cabeçalho do próprio arquivo, sendo que o mesmo só é liberado para leitura ou escrita pelo aplicativo, com a informação da senha, na hora de sua abertura.

3 MECANISMO DE SENHA IMPLEMENTADO NO NÚCLEO DO SISTEMA OPERACIONAL LINUX – KERNEL VERSÃO 2.4.0

O mecanismo descrito a seguir foi implementado para o kernel versão **2.4.0**, sistema de arquivos **ext2**, e não há garantias de que funcione em versões anteriores ou posteriores, salvo adaptações para cada versão em particular.

O mecanismo consiste da adoção de uma senha criptografada para cada arquivo em disco. Essa senha é salva no cabeçalho (inode) do arquivo e com isso poderemos controlar a abertura de tal arquivo e até mesmo criptografá-lo através desta senha, controlando assim seu uso, bem como a visualização de seu conteúdo.

A idéia original baseava-se no acréscimo de um quarto parâmetro na chamada de função **open**, responsável pela abertura e ou criação de arquivos.

- o protótipo da função original:

asmlinkage long sys_open(const char *, int, int);

- seria substituído por:

asmlinkage long sys_open(const char *, int, int, const char *);

- e internamente substituiríamos sua chamada de sistema para que reconheça um parâmetro adicional (parâmetro denominado password). A chamada seria a **_syscall4** que aceita quatro parâmetros:

static inline _syscall4(int, open, const char *, file, int, flag, int, mode, const char *, password)

```
#define _syscall4(type, name, type1, arg1, type2, arg2, type3, arg3, type4, arg4)
type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4)
{
        long __res;
        __asm__ volatile ("int $0x80"
```

```
                : "=a" (__res)
                : "0" (__NR_##name),"b" ((long)(arg1)), "c" ((long)(arg2)), "d"
                ((long)(arg3)), "S" ((long)(arg4)));
        __syscall_return(type,__res);
}
```

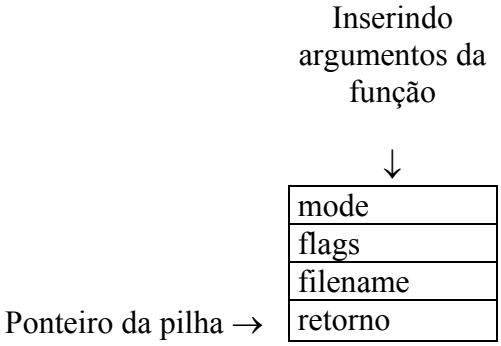
- o retorno da função é carregado no registrador **eax** e seus argumentos, a saber: file em **ebx**, flag em **ecx**, mode em **edx** e, por fim, password em **esi,** seguindo-se com a chamada do serviço **0x80**, que é o gancho para as funções básicas com que o kernel nos provê. O retorno da chamada é dado em **eax**. As devidas modificações para que tal mecanismo funcione são internas e não necessitariam de mais mudanças na função **open**. Assim, todos os aplicativos se beneficiariam deste novo recurso.

Entretanto, tal solução não seria viável na prática, por um motivo muito simples: os parâmetros formais desta função modificada não seriam compatíveis em número com todos os aplicativos que utilizam a função **open** e até mesmo com os drives de dispositivos e utilitários do próprio kernel, que também não estariam preparados para mais um parâmetro desta função modificada.

Se uma função usa argumentos, ela deve declarar variáveis que aceitem os valores dos argumentos. Essas variáveis são chamadas de parâmetros formais da função. Elas se comportam como quaisquer variáveis locais dentro da função e são criadas na entrada e destruídas na saída da função. Uma região de memória conhecida como pilha é utilizada para a passagem dos parâmetros formais na chamada de uma função. A própria função retira os valores armazenados na pilha (dizemos desempilha).

O ponteiro da pilha sempre aponta para uma determinada região chamada de topo da pilha. Quando colocamos uma palavra qualquer na pilha, o ponteiro da pilha é decrementado, afim de apontar o valor anterior. Desta maneira, empilhamos um valor sobre a pilha. Se desempilhamos um valor, esse será o último valor empilhado, ou seja, o ponteiro da pilha se movimenta na ordem reversa ao empilhamento.

Ainda referindo-se a esses aplicativos, em termos práticos, uma chamada à função **open** por qualquer que seja o aplicativo, seria realizada da seguinte forma:



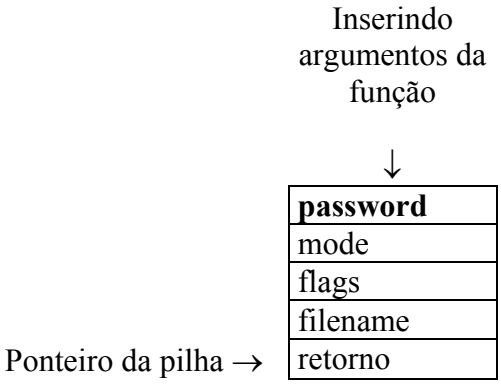
Obs.: A convenção de chamada de C difere das outras linguagens de alto nível, porque empilha os valores a serem passados em ordem inversa (da direita para à esquerda). Uma forma comum e útil de imaginar a pilha é pensar na abreviatura FILO, que significa **F**irst in, **L**ast **O**ut (primeiro a entrar, último à sair).

- desempilhando esses valores com a função **open** modificada, teríamos:

```
movl   $__res, %eax      // carrega eax com o retorno da chamada
movl   $filename, %ebx   // carrega ebx com o nome do arquivo
movl   $flags, %ecx      // carrega ecx com os flags
movl   $mode, %edx       // carrega edx com o modo
movl   $password, %esi   // carrega esi com a senha
int    $0x80             // executa a chamada de sistema
```

- e logicamente teríamos resultados inesperados, pois, desempilhamos um provável quarto valor empilhado e carregamos o registrador **esi** com este valor. Entretanto, estamos abordando a situação de um aplicativo que utiliza uma chamada **open** sem a devida modificação, e o quarto parâmetro não foi por ele fornecido. Sendo assim, carregamos o registrador **esi** com valores inconsistentes ou valores fragmentados de memória.

Para que tudo funcione a contento, precisaríamos fornecer para esta função **open** modificada, um necessário quarto parâmetro:



desempilhando esses valores:

```
movl   $__res, %eax
movl   $filename, %ebx
movl   $flags, %ecx
movl   $mode, %edx
movl   $password, %esi
int    $0x80
```

- teríamos neste caso uma igualdade de parâmetros e o funcionamento esperado, já que desempilhamos um valor consistente (fornecido pela chamada da função) para carregarmos o registrador **esi**.

Para se resolver este impasse, teríamos alternativas se utilizássemos na compilação do kernel do linux o C++, que é uma linguagem mais moderna e utiliza recursos como sobrecarga de funções e valores default na passagem de parâmetros:

- sobrecarga de função (funções que executam tarefas semelhantes e que tem o mesmo nome, e são distinguidas pelo sistema pelo tipo e quantidade de seus parâmetros, neste caso sobrecarregamos a função **open** original, por uma outra função de mesmo nome e com um parâmetro adicional: o parâmetro password):

asmlinkage long sys_open(const char * filename, int flags, int mode);

asmlinkage long sys_open(const char * filename, int flags, int mode, const char * password);

- valores default para passagem de parâmetros, se a função for chamada sem esse parâmetro, o valor default é assumido (os argumentos com valor default, têm que necessariamente ser os últimos argumentos da função. Neste caso o parâmetro **password** é declarado como default e caso não seja fornecido por uma chamada da função **open**, o valor padrão é assumido como **nulo**):

asmlinkage long sys_open(const char * filename, int flags, int mode);

asmlinkage long sys_open(const char * filename, int flags, int mode, const char * password = NULL);

O compilador utilizado para compilar o kernel do sistema linux é o famoso gcc, e é provável que uma versão mais moderna possa trazer recursos como sobrecarga de função e passagem de parâmetros com valores default (implementados para o ambiente C), mas optamos por inserir no kernel do linux uma nova função para abertura e ou criação de arquivos, denominada **xopen**, que além dos parâmetros convencionais, foi adicionado outro parâmetro que é a senha, necessária à criação ou abertura de arquivos:

asmlinkage long sys_xopen(const char * filename, int flags, int mode, const char * password);

Assim, esta versão modificada do kernel 2.4.0 não se torna incompatível com os aplicativos já existentes e podemos utilizar esse novo recurso sem nenhum problema. Basta que também modifiquemos um compilador qualquer, para que reconheça e possa utilizar esse novo recurso quando da abertura e ou criação de arquivos. É importante justificar que os arquivos criados sob esta versão modificada de kernel, podem ser abertos em outros sistemas linux, já que não estão preparados para reconhecer uma senha armazenada no inode destes arquivos. A diferença é que esta senha pode ser utilizada para criptografar tal arquivo e, mesmo aberto por outro sistema, seu conteúdo está protegido via criptografia e não será inteligível.

Ao analisarmos o inode de arquivos para o sistema **ext2**, podemos encontrar um campo reservado que não é utilizado pelo kernel versão **2.4.0**. Esse campo é **l_i_reserved1**, do tipo inteiro longo (32 bits). Esse campo é utilizado para armazenar a senha fornecida pelo parâmetro **password** da função **xopen**. Para que possamos salvá-la no inode, quando da criação de um arquivo, ou compará-la com uma senha já armazenada no inode quando da abertura de um arquivo, tudo o que precisamos é transformar essa senha que é uma constante de caracteres em um tipo compatível a **l_i_reserved1**. Podemos usar os recursos de criptografia do próprio linux, e com isso podemos armazenar esta senha criptografada, dando proteção total a este mecanismo implementado. Acessamos este campo através de **osd1.linux1.l_i_reserved1**, da estrutura **ext2_inode**.

```
/*
 * estrutura do inode de disco (sistema de arquivos ext2)
 */

struct ext2_inode {
  __u16  i_mode;        /* File mode */
  __u16  i_uid;         /* Low 16 bits of Owner Uid */
  __u32  i_size;        /* Size in bytes */
  __u32  i_atime;       /* Access time */
  __u32  i_ctime;       /* Creation time */
  __u32  i_mtime;       /* Modification time */
```

```
  __u32  i_dtime;       /* Deletion Time */
  __u16  i_gid;         /* Low 16 bits of Group Id */
  __u16  i_links_count;  /* Links count */
  __u32  i_blocks;      /* Blocks count */
  __u32  i_flags;       /* File flags */
  union {
    struct { __u32  l_i_reserved1; } linux1;
    struct { __u32  h_i_translator; } hurd1;
    struct { __u32  m_i_reserved1; } masix1;
  } osd1;                           /* OS dependent 1 */
  __u32  i_block[EXT2_N_BLOCKS];          /* Pointers to blocks */
  __u32  i_generation;                /* File version (for NFS) */
  __u32  i_file_acl;                 /* File ACL */
  __u32  i_dir_acl;                  /* Directory ACL */
  __u32  i_faddr;                    /* Fragment address */
  union {
    struct {
      __u8  l_i_frag;               /* Fragment number */
      __u8  l_i_fsize;              /* Fragment size */
      __u16  i_pad1;
      __u16  l_i_uid_high;              /* these 2 fields    */
      __u16  l_i_gid_high;              /* were reserved2[0] */
      __u32  l_i_reserved2;
    } linux2;
    struct {
      __u8  h_i_frag;               /* Fragment number */
      __u8  h_i_fsize;              /* Fragment size */
      __u16  h_i_mode_high;
      __u16  h_i_uid_high;
      __u16  h_i_gid_high;
      __u32  h_i_author;
    } hurd2;
    struct {
      __u8  m_i_frag;               /* Fragment number */
      __u8  m_i_fsize;              /* Fragment size */
      __u16  m_pad1;
      __u32  m_i_reserved2[2];
    } masix2;
  } osd2;                           /* OS dependent 2 */
};
```

No kernel do linux, quando da abertura e ou criação de arquivos, precisamos encontrar em que ponto podemos trabalhar diretamente no inode desses arquivos. Esse ponto é encontrado na função **open_namei**, presente no arquivo **namei.c**. Ela é chamada pela função **filp_open** que por sua vez é chamada por **xopen** (presentes no

arquivo **open.c**), e mesmo nesta função ainda não temos acesso direto ao inode de disco. Tudo o que temos é o acesso ao inode de memória e é nele que temos que trabalhar para que tenhamos a transferência dessa senha para o inode de disco, quando estamos criando um arquivo e do inode de disco para o inode de memória, quando abrimos um arquivo. No inode de memória, encontramos uma região denominada **ext2_i** que aponta para a estrutura **ext2_inode_info**. Esta estrutura guarda informações sobre o inode de disco e é nela que encontramos outro campo reservado, não utilizado por esta versão do kernel (**2.4.0**). Este campo, denominado **not_used_1**, é utilizado para armazenar temporariamente a senha. Quando abrimos o arquivo, a função **ext2_read_inode** é encarregada de ler o inode de disco e transferir o valor salvo no campo **osd1.linux1.l_i_reserved1** para o inode de memória (campo **u.ext2_i.not_used_1**) e quando criamos um arquivo a função **ext2_update_inode**, transfere a senha armazenada no campo **u.ext2_i.not_used_1** do inode de memória para o campo **osd1.linux1.l_i_reserved1** do inode de disco. As funções **ext2_read_inode** e **ext2_update_inode** estão presentes no arquivo **inode.c**. Este arquivo é dependente do sistema de arquivos e se encontra no diretório específico do sistema de arquivos ext2.

```
/*
 estrutura do inode de memória que guarda informações sobre o inode de disco do
sistema de arquivos ext2
*/

struct ext2_inode_info {
  __u32  i_data[15];
  __u32  i_flags;
  __u32  i_faddr;
  __u8   i_frag_no;
  __u8   i_frag_size;
  __u16  i_osync;
  __u32  i_file_acl;
  __u32  i_dir_acl;
  __u32  i_dtime;
  __u32  not_used_1;          /* FIX: not used/ 2.2 placeholder */
  __u32  i_block_group;
  __u32  i_next_alloc_block;
  __u32  i_next_alloc_goal;
  __u32  i_prealloc_block;
  __u32  i_prealloc_count;
  __u32  i_high_size;
```

```
  int    i_new_inode:1;       /* Is a freshly allocated inode */
};


/*
 * estrutura do inode de memória
 */


struct inode {
  struct list_head  i_hash;
  struct list_head  i_list;
  struct list_head  i_dentry;

  struct list_head  i_dirty_buffers;

  unsigned long     i_ino;
  atomic_t          i_count;
  kdev_t            i_dev;
  umode_t           i_mode;
  nlink_t           i_nlink;
  uid_t             i_uid;
  gid_t             i_gid;
  kdev_t            i_rdev;
  loff_t            i_size;
  time_t            i_atime;
  time_t            i_mtime;
  time_t            i_ctime;
  unsigned long     i_blksize;
  unsigned long     i_blocks;
  unsigned long     i_version;
  struct semaphore  i_sem;
  struct semaphore  i_zombie;
  struct inode_operations *i_op;
  struct file_operations  *i_fop;  /* former ->i_op->default_file_ops */
  struct super_block      *i_sb;
  wait_queue_head_t       i_wait;
  struct file_lock        *i_flock;
  struct address_space    *i_mapping;
  struct address_space    i_data;
  struct dquot            *i_dquot[MAXQUOTAS];
  struct pipe_inode_info  *i_pipe;
  struct block_device     *i_bdev;

  unsigned long           i_dnotify_mask; /* Directory notify events */
  struct dnotify_struct   *i_dnotify;      /* for dir. notifications */

  unsigned long           i_state;
```

```
 unsigned int          i_flags;
 unsigned char          i_sock;

 atomic_t              i_writecount;
 unsigned int          i_attr_flags;
 __u32                 i_generation;
 union {
  struct minix_inode_info      minix_i;
  struct ext2_inode_info       ext2_i;
  struct hpfs_inode_info       hpfs_i;
  struct ntfs_inode_info       ntfs_i;
  struct msdos_inode_info      msdos_i;
  struct umsdos_inode_info     umsdos_i;
  struct iso_inode_info        isofs_i;
  struct nfs_inode_info        nfs_i;
  struct sysv_inode_info       sysv_i;
  struct affs_inode_info       affs_i;
  struct ufs_inode_info        ufs_i;
  struct efs_inode_info        efs_i;
  struct romfs_inode_info      romfs_i;
  struct shmem_inode_info      shmem_i;
  struct coda_inode_info       coda_i;
  struct smb_inode_info        smbfs_i;
  struct hfs_inode_info        hfs_i;
  struct adfs_inode_info       adfs_i;
  struct qnx4_inode_info       qnx4_i;
  struct bfs_inode_info        bfs_i;
  struct udf_inode_info        udf_i;
  struct ncp_inode_info        ncpfs_i;
  struct proc_inode_info       proc_i;
  struct socket                socket_i;
  struct usbdev_inode_info     usbdev_i;
  void                         *generic_ip;
 } u;
};
```

Internamente a função **xopen** é uma réplica da função **open**, exceto pela quantidade de parâmetros. As duas chamam a função **filp_open** que foi modificada para receber mais um parâmetro (a senha). Todas as chamadas internas à função **filp_open** realizadas pelo kernel do linux foram modificadas, atribuindo um valor nulo para o parâmetro password, já que é necessário. A função **filp_open** chama a função modificada **open_namei**, transferindo o parâmetro **password,** que pode receber um argumento no formato de cadeia de caractere (a senha). É na função **open_namei** que

temos as modificações reais, incluindo código para criptografar a senha e em seguida convertê-la para um tipo inteiro longo compatível com o campo **u.ext2_i.not_used_1**. Utilizamos um algoritmo denominado Hash para converter a cadeia de caracteres da senha no tipo inteiro longo, necessário para o armazenamento no campo correspondente na estrutura do inode, o algoritmo hash se encontra em um arquivo denominado **Hash.c**.

A função **open_namei** neste caso é utilizada em duas situações, na primeira é encarregada de comparar a senha fornecida através do parâmetro password com a senha salva no inode de disco, abortando a operação de leitura do arquivo, caso a senha fornecida não seja correspondente. Na segunda é tratada a criação de um arquivo onde a senha é salva no inode de memória para posterior transferência ao inode de disco.

Esta transferência, quando da criação de um arquivo, ocorre na função **ext2_update_inode** e, quando da abertura de um arquivo, ocorre na função **ext2_read_inode.** Estas funções se encontram no arquivo **inode.c,** especificamente no diretório responsável pela manipulação do sistema de arquivo Ext2.

4. CONCLUSÃO

A necessidade de proteção de informações gera hoje uma grande preocupação em todo o segmento da informática. Grandes investimentos estão sendo aplicados para o desenvolvimento de tecnologia para garantir o direito do usuário sobre suas informações.

Com o constante aumento de informações geradas e guardadas, e o crescimento das redes de computadores, torna-se cada vez mais difícil cuidar do sigilo das informações.

Este trabalho introduziu a proteção de arquivos gerenciada pelo próprio sistema operacional, cuja proposta visa assegurar que o proprietário do arquivo tenha acesso a suas informações com confiança. Não caracterizado a permissão ao acesso de arquivos ao usuário logado na máquina, mas sim ao proprietário, pois o sistema operacional fará a solicitação da senha do arquivo e não a senha que o usuário utiliza para conexão na rede em que está trabalhando.

Isto proporciona uma maior segurança, pois se um arquivo for copiado por uma pessoa não autorizada, a mesma ficará sem os dados do arquivo, pois o sistema operacional solicitará a senha do proprietário para abertura do mesmo.

Para implementação deste trabalho, foi necessário inserir um quarto parâmetro na chamada de abertura e criação de arquivos no sistema operacional. Isto foi possível utilizando um campo reservado do inode.

Para que isto possa funcionar corretamente, será necessário a atualizar os aplicativos existentes no mercado, pela inclusão de uma instrução de solicitação, para que o mesmo aceite a inserção de um quarto parâmetro na estrutura de cabeçalho de seus arquivos.

Esta solução poderá ser utilizada apenas em sistema operacional com versão atualizada após a conclusão deste trabalho. Os sistemas anteriores a este não estarão habilitados para solicitação desta senha, pois seu núcleo não conterá a rotina de

solicitação do quarto parâmetro, que é a senha. Mesmo assim, os arquivos serão utilizados sem nenhum problema, pois o sistema operacional ignorará este parâmetro na abertura ou criação de arquivo.

REFERÊNCIAS BIBLIOGRÁFICAS

CROWLEY, Charles. *Operating systems: a design-oriented approach*. USA : Times Mirror Higher Education Group. 1997.

DEITEL, Harvey M. "*An introduction to operating systems*".2nd. Boston College. USA : Addison-Wesley Publishing Company, Inc. 1990.

GARFINKEL, Simson & SPAFFORD, Gene. *Practical UNIX and internet security*. 2.ed. USA, Sabastopol : O'Reilly & Associates. Inc. 1996.

LANCHARRO, Eduardo Alcalde; PASCUAL, Juan Morera; ATANASIO, Juan A. Perez-Campanero. *Introdução aos sistemas operativos (Ms/Dos, Unix, OS/2, MVS, VMS, OS/400)*. Trad. AVELAR, Pedro Roovers de e FRAGOSO, Augusto M. M. P.. Lisboa : McGraw-Hill de Portugal. 1991.

MACHADO, Francis B. e MAIA, Luiz Paulo. *Introdução à arquitetura de sistemas operacionais*. Rio de Janeiro : LTC. 1995. p. 25

_____. *Arquitetura de sistemas operacionais*. Rio de Janeiro : LTC, 2000. p. 27.

MINASI, Mark et al. *OS/22.1 Guia Completo*. Trad. Angélica Liomar Soares de Moura. Rio de Janeiro : Berkeley, 1994.

NORTON, Peter. *A bíblia do programador: a referência mais completa para o IBM PC, computadores compatíveis a softwares básicos*. Trad. Geraldo Costa Filho. Rio de Janeiro : Campus, 1994.

SILBERSCHATZ, Abraham; GALVIN, Peter. *Operating system concepts*. 5.ed. USA : Addison Wesley Longman, Inc.

SHAY, William A. *Sistemas operacionais*. Trad. Mário Moro Fechio. São Paulo : Makron Books, 1996.

TANENBAUM, Andrew S. & WOODHULL, Albert S.. *Sistemas operacionais: projeto e implementação*. Trad. Edson Furmankiewicz. 2. Ed. Porto Alegre : Bookman, 2000.

THOMAS, Rebecca; YATES, Jean. *Unix guia do usuário*. Trad. Maria Cláudia de Oliveira Santos. São Paulo : McGraw-Hill, 1989.

http://www.pwr.com.br/servicos/respostas_el_37-46.htm

http://br.tldp.org/documentos/livros/html/gas/node44.html, 2000

ANEXO

ANEXO 01

ALTERAÇÕES EFETUADAS NO COMPILADOR FREE PASCAL

ALTERAÇÕES EFETUADAS NO COMPILADOR FREE PASCAL

Para testarmos nosso mecanismo de proteção de arquivos por senha armazenada diretamente no inode do arquivo, precisamos de um compilador que possa gerar executáveis que tirem proveito desse novo recurso, e como não havia nada disponível, partimos para a modificação de um compilador para que tal tarefa fosse cumprida. Mas para isso precisamos de um compilador que tenha o código fonte disponível, que seja fácil para implementar tais mudanças e que também seja confiável. Encontramos o Free Pascal que oferece todas as possibilidades para que tais modificações fossem implementadas: a não dependência de bibliotecas específicas e a organização de sua rtl (run time library) facilitaram as mudanças e o resultado esperado foi atingido. A versão utilizada foi a 0.99.14 e caso seja necessário implementar esses recursos em outras versões, as modificações serão mínimas.

Segue um resumo das principais modificações e as unidades onde ocorreram:

Unit filutil.inc — Implementação das funções sobrecarregadas FileOpen e FileCreate, declaradas na unidade filutilh.inc. Essas funções contêm um parâmetro adicional (password).

Unit filutilh.inc — As funções originais (FileOpen e FileCreate) foram sobrecarregadas por duas similares, mas acrescentando às similares um parâmetro adicional do tipo cadeia de caracteres (password). Nesta unidade foram alterados apenas os protótipos das funções.

Unit objinc.inc — Foi sobrecarregada a função FileOpen, acrescentando mais um parâmetro (password), do tipo cadeia de caracteres, na função sobrecarregada esse parâmetro foi repassado à função SYS_open. (A função FileOpen, declarada nesta unidade, difere da função FileOpen declarada na unidade filutil.inc, no tipo de argumentos e no tipo de valor de retorno).

Unit errors.pp — Foi acrescentado um novo código de erro, número 125 com a mensagem: Invalid Password.

Unit syscalls.inc — Foi modificada a função SYS_open acrescentando-se a ela mais um parâmetro (password). Esta função é a responsável direta pela chamada da função **open** do linux. Como temos duas funções para abertura e ou criação de arquivos exportadas pelo linux (open e xopen), chamamos uma ou outra de acordo com o valor do parâmetro password recebido pela função SYS_open. E tivermos um valor nulo chamamos open (função de nº 5), caso tenhamos um valor para o parâmetro password, chamamos a função xopen (função de nº 222).

A função OpenDir, recebeu também uma modificação, adicionado um valor nulo para o parâmetro password, necessário à função SYS_open, chamada por ela.

Unit sysnr.inc — Foi acrescentado um valor constante, número 222 (syscall_nr_xopen), para que fosse possível reconhecer a nova função xopen exportada pelo linux e que tem esse número. Essa constante é utilizada pela função SYS_open, que executa uma chamada direta às funções para criação e abertura de arquivos no linux (open e xopen).

Unit syslinux.pp — O procedimento Do_Open, teve que ser modificado, pois o mesmo executa uma chamada à função SYS_open, que necessita de mais um parâmetro (password). Foi utilizado um argumento de valor nulo para a chamada da função SYS_open.

Unit linux.pp — Foi adicionada uma nova função fdOpen, sobrecarregando a função fdOpen original. Esta nova função foi adicionada de mais um parâmetro (password), a que pode ser atribuído um valor do tipo cadeia de caracteres ou ponteiro para caracteres.

ANEXO 02

ARQUIVOS ALTERADOS NO KERNEL DO LINUX, VERSÃO 2.4.0

ARQUIVOS ALTERADOS NO KERNEL DO LINUX, VERSÃO 2.4.0

	DIRETÓRIO	ALTERAÇÃO
Makefile	/fs	Inclusão do arquivo **hash.o**, para efeito de automatização da compilação.
dquot.c	/fs	Alteração da chamada da função **filp_open**, acrescentando a ela um parâmetro nulo como senha. (Utilizado na função **quota_on**).
exec.c	/fs	Alteração da chamada da função **filp_open**, acrescentando a ela um parâmetro nulo como senha. (Utilizado na função **do_coredump**).
hash.c	/fs	Definição da função **hash**, utilizada pelo sistema para converter cadeias alfanuméricas em números inteiros longos.
namei.c	/fs	Alteração da função **open_namei**. Nesta função está implementado o mecanismo de validação da senha de arquivos. Basicamente todo o procedimento está relacionado ao recebimento do novo parâmetro "**password**".
open.c	/fs	Alteração da função **filp_open**, acrescentando um novo parâmetro do tipo cadeia de caracter "**password**", parâmetro repassado à função **open_namei**, presente no arquivo **namei.c**. Alteração da função **sys_open**, especificamente na chamada da função **filp_open**, acrescentado a ela um parâmetro nulo como senha. Inclusão da função **sys_xopen**; esta função é similar a função **sys_open**, difere apenas na chamada da função **filp_open**, repassando a ela o novo parâmetro "**password**".

inode.c	/fs/ext2	Alteração na função **ext2_read_inode**, responsável pela leitura do inode de disco (lê o campo **`osd1.linux1.l_i_reserved1`** do inode de disco e transfere para o campo **`u.ext2_i.not_used_1`**, do inode de memória). Alteração na função **ext2_update_inode**, responsável pela atualização do inode de disco (transfere o conteúdo do campo **`u.ext2_i.not_used_1`** de inode de memória para o campo **`osd1.linux1.l_i_reserved1`** do inode de disco).
acct.c	/kernel	Alteração da chamada da função **filp_open**, acrescentando a ela um parâmetro nulo como senha. (Utilizado na função **sys_acct**).
security.c	/net/khttpd	Alteração da chamada da função **filp_open**, acrescentando a ela um parâmetro nulo como senha. (Utilizado na função **OpenFileForSecurity**).
errno.h	/include/asm-i386 (Arquivo específico para arquitetura 386).	Foi acrescentado um novo número de erro: **125** "**`EBADPWD`**", senha requerida ou inválida.
unistd.h	/include/asm-i386 (Arquivo específico para arquitetura 386).	Alterada a definição de **`__syscall_return`**, para que o novo número de erro (**125**) seja reportado pelo kernel, também foi acrescentado uma nova definição "**`__NR_xopen`**" número **222**, no rol de chamadas de sistema exportado pelo kernel. Acrescentada a função **xopen** do tipo **`static inline`**.
fs.h	/include/kernel	Protótipos das funções alteradas **filp_open**, **open_namei** e da função acrescentada **sys_xopen**.
hash.h	/include/kernel	Protótipo da nova função **hash**.