Paulo Sérgio Rezende de Carvalho

Desenvolvimento de um Cliente PAM para Autenticação Criptografada e Distribuída de Estações Linux

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências da disciplina Projeto Orientado para obtênção do título de Bacharel em Ciência da Computação.

Orientador
Prof. Joaquim Quinteiro Uchôa

Lavras
Minas Gerais - Brasil
2002

Paulo Sérgio Rezende de Carvalho

Desenvolvimento de um Cliente PAM para Autenticação Criptografada e Distribuída de Estações Linux

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências da disciplina Projeto Orientado para obtênção do título de Bacharel em Ciência da Computação.

Aprovada em *09 de agosto de 2002*

Prof. Bruno de Oliveira Schneider

Prof. Joaquim Quinteiro Uchôa
(Orientador)

Lavras
Minas Gerais - Brasil

A
minha querida mãe,
que sempre me incentivou
e acreditou nas minhas
loucuras.

Agradecimentos

Agradeço aos meus pais, Paulo e Terezinha.

Agradeço aos meus irmãos, Maga, Lili e Júlio Cesar.

Agradeço aos meus amigos de Itaúna, Flávio e Edd.

Agradeço ao pessoal do apartamento 201, Eduardo, Alexandre, André, Júlio, Wanner, Talles e Mário.

Agradeço aos meus amigos de turma Alisson, Alessandra, Deive, Giselle, Gláucia, Joseane e Marcos Paulo.

Agradeço aos meus amigos da Comp., Mário Luis, Jones, Flávio Diego, Cícero, Renata, Rodrigo e Lívia.

Resumo

Este trabalho tem por objetivo a construção de um cliente PAM para autenticação criptografada e distribuída em estações Linux. O atual sistema de autenticação valida um usuário através da comparação de sua senha com a senha presente em um arquivo na máquina local. Afim de eliminar esta autenticação local foi implementado um aplicativo cliente e um aplicativo servidor, para estabelecer a comunicação entre o usuário e o servidor de senhas, agora o arquivo de senhas ficará apenas no servidor e não mais nas máquinas locais. O aplicativo capaz de autenticar um usuário remotamente está presente no servidor de senhas e este utiliza módulos *PAM* para efetuar esta autenticação e SSL para a comunicação segura. Além de autenticar um usuário, o usuário tem o poder de alterar sua senha quando for de seu interesse. Como o próprio nome sugere, o aplicativo cliente está presente nas estações, enquanto que o aplicativo servidor está presente no servidor de senhas Linux.

Sumário

1 Introdução **1**

2 Autenticação Linux via PAM **3**
2.1 Visão Geral do PAM 3
2.2 Configuração do PAM 5
2.3 Módulos PAM Disponíveis 7
2.4 Desenvolvimento de Módulos PAM 8
2.4.1 Apontando Dados 9
2.4.2 Pegando Dados 10
2.4.3 Apontando Itens 10
2.4.4 Pegando Itens 10
2.4.5 Obtendo o Nome de Usuário 11
2.4.6 Gerenciamento de Autenticação 11

3 Comunicação Segura **13**
3.1 Criptografia 13
3.1.1 Conceitos e Históricos 13
3.1.2 O propósito da Criptografia 15
3.1.3 Exemplos de Métodos Criptográficos 15
3.1.4 Senhas no Linux 19
3.2 Sockets 20
3.2.1 Compreendendo uma Conexão de Socket 20
3.2.2 As Principais Chamadas Sockets 21
3.2.3 Uma Típica Conversação Cliente/Servidor 23
3.3 SSL (*Secure Sockets Layer*) 25
3.3.1 Camadas 26
3.3.2 Comentários Finais 27

4 Como é Feita a Autenticação em Linux **29**
4.1 LDAP 30
4.1.1 Serviço de Diretório 30
4.1.2 Acessando a Informação 31
4.1.3 Funcionamento do LDAP 31
4.2 NIS 31
4.3 Radius/PortSlave 32
4.4 Comentários Finais 33

5 Proposta de Trabalho **35**
5.1 Solução Proposta 35
5.2 Estratégia de Implementação 36
5.2.1 Características da Implementação 37
5.3 Instalação e Configuração 38

6 Considerações Finais **43**

Lista de Figuras

2.1 Funcionamento do PAM 4

3.1 Criptografar e Descriptografar 14
3.2 Criptografia Convencional ou Criptografia de Chave Simétrica . . 16
3.3 Criptografia de Chave Pública 17
3.4 Codificação Utilizando uma Função Hash 19
3.5 Fluxo da Transação de Sockets 24

5.1 Esquema de Comunicação Estações/Servidor 36
5.2 Usuário Requisitando Mudança de Senha 41

Lista de Tabelas

2.1 Conteúdo do Diretório `/etc/pam.d` 5
2.2 Configuração do `passwd` 6
2.3 Tipos de Modulos PAM 6
2.4 Flags de Controle 7
2.5 Argumentos Padrão 8
2.6 Módulos PAM disponíveis 8

5.1 Arquivo de Configuração `senhapam` 39

Capítulo 1

Introdução

Um problema bastante usual em laboratórios de computação é o de criar um sistema unificado de autenticação. Esse sistema é responsável por garantir que todos os usuários desse laboratório possam ter suas contas de acesso cadastradas em um único servidor.

Com a autenticação unificada, um conjunto de máquinas compartilham informações diversas para autenticar e validar a identidade dos usuários, facilitando o trabalho de usuários e administradores de um sistema distribuído.

Em Linux, a situação não é diferente: comumente, em listas de discussão sobre Linux, surgem questões sobre como fazer uma autenticação unificada em laboratórios de computação ou sobre qual a melhor forma de fazê-lo.

Originalmente a autenticação no Linux era apenas via comparação de senhas criptografadas armazenadas em um arquivo local chamado `/etc/passwd`. Um programa como o *login* pedia o nome do usuário e a senha, então criptografava a senha e comparava o resultado com o armazenado naquele arquivo. Se fossem iguais, garantia o acesso à máquina. Caso contrário, retornava erro de autenticação. Isto até funciona muito bem para o programa *login*, mas, digamos que agora eu queira usar isso também para autenticação remota. Ou seja, a base de usuários não está mais na mesma máquina, mas sim em alguma outra máquina da rede, o chamado servidor de autenticação. Teremos que mudar o programa *login* para que ele também suporte esse tipo de autenticação remota.

Surgiu um novo algoritmo de criptografia, muito mais avançado, mais rápido, criptografa melhor, etc. Queremos usar esse novo algoritmo. Claro que teremos que mudar novamente o programa *login* para que ele suporte este novo algoritmo também. No Linux, muitos programas utilizam algum tipo de autenticação de

usuários. Imagine se todos eles tivessem que ser reescritos cada vez que se mudasse algum dos critérios de autenticação.

Para resolver este tipo de problema, a Sun® criou o PAM em 1995 [Sme95] e liberou as especificações em forma de RFC (*Request for Comment*) [Sun01]. O Linux derivou sua implementação do PAM a partir deste documento. Com PAM, o aplicativo *login* deste exemplo teria que ser reescrito apenas uma vez, justamente para suportar PAM. A partir de então, o aplicativo delega a responsabilidade da autenticação para o PAM e não se envolve mais com isso.

PAM é a parte principal da autenticação em um sistema Linux. PAM significa *Pluggable Authentication Modules*, ou Módulos de Autenticação Plugáveis.

Este trabalho tem o objetivo de criar um cliente *PAM* para autenticar e modificar a senha de usuários remotamente. Desta forma o arquivo de senhas dos usuários fica em um servidor, o servidor de senhas, e toda vez que um usuário se conectar a uma estação, este usuário estará enviando uma requisição de autenticação ao servidor de senhas. Se o usuário for um usuário legítimo, então o servidor devolve uma mensagem para estação, liberando-a para este usuário, caso contrário devolve um erro de autenticação. Para estabelecer a comunicação entre a estação e o servidor de senhas criou-se uma conexão via SSL *Secure Sockets Layer* [Neta].

Capítulo 2

Autenticação Linux via PAM

PAM é a parte principal da autenticação em um sistema Linux. PAM significa *Pluggable Authentication Modules*, ou Módulos de Autenticação Plugáveis.

Embora o PAM não possa proteger seu sistema após este ter sido violado, ele pode certamente ajudá-lo a prevenir seu sistema de um ataque. Ele faz isso através de um forte esquema de configuração. Por exemplo, convencionalmente usuários UNIX autenticam-se fornecendo uma senha ao *prompt* de senha após ele ter digitado seu nome de usuário ao *prompt* de *login*. Em muitas circunstâncias, como acesso interno às estações de trabalho, esta simples forma de autenticação é suficiente. Em outros casos, mais informações são necessárias. Se um usuário quer se conectar a um sistema interno de uma origem externa, como a Internet, mais informações devem ser requisitadas. PAM provê de módulos que permitem configurar seu ambiente com o devido grau de segurança.

2.1 Visão Geral do PAM

A Figura 2.1 apresenta uma visão geral do diagrama do linux-PAM em interação com aplicações Linux. Este diagrama descreve os principais componentes de uma implementação PAM - aplicações como `login, ftp, su` etc.

A parte do Linux-PAM (bibliotecas PAM, encontradas em `/lib`), responsável pelo carregamento dos módulos PAM necessários para o funcionamento das aplicações é baseada nos arquivos de configuração, encontrados em `/etc/pam.d`. Geralmente segue-se o fluxo de execução:

1. A aplicação - por exemplo `login` - faz uma chamada ao Linux-PAM.

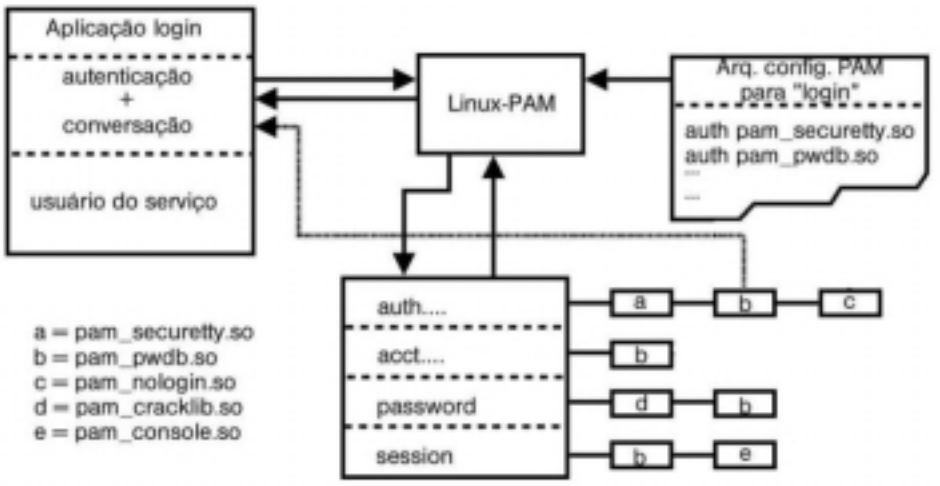

Figura 2.1: Funcionamento do PAM

2. Linux-PAM localiza a configuração apropriada no diretório `/etc/pam.d` (ou, alternativamente, no arquivo `/etc/pam.conf`) para obter a lista dos módulos necessários para o serviço requerido.

3. Linux-PAM então carrega cada módulo na ordem dada pelo arquivo de configuração para processar. Dependendo da configuração dos parâmetros, nem todos os módulos listados no arquivo de configuração serão necessariamente executados.

4. Alguns, ou todos os módulos devem *conversar* com o usuário através da aplicação. Esta conversação normalmente inclui algumas informações que estão presentes no *prompting* do usuário, como o pedido de senha, e este recebe uma resposta. Se a resposta do usuário satisfizer um módulo PAM em particular, ou se este módulo for satisfeito por alguma maneira, o controle é passado de volta ao Linux-PAM para que este processe o próximo módulo (os passos 3 e 4 são repetidos para cada módulo relatado no arquivo de configuração, que esteja associado a aplicação em questão). No final das contas, o processo termina com um sucesso ou falha. No caso de falha, normalmente a menssagem apresentada ao usuário não indica a causa da falha. Esta mensagem geral é uma forma de garantir a segurança do sistema.

Porém, a maioria dos módulos PAM, dispõem diversos níveis de *logs*, permitindo ao administrador do sistema rastrear os problemas e identificar as violações de segurança.

2.2 Configuração do PAM

Existem duas configurações diferentes para o PAM. A primeira induz o PAM a utilizar uma das configurações presentes em um arquivo único, o `pam.conf`, presente no diretório `/etc` ou olhar a coleção de arquivos de configuração presentes no diretório `/etc/pam.d`, mas não em ambos. A segunda opção usa ambos mecanismos, e as entradas do diretório `/etc/pam.d` sobrepõem as do arquivo`/etc/pam.conf`. A primeira opção é recomendada e reflete a implementação usada nas distribuições Red Hat [MM00a].

Existe uma pequena diferença entre usar o arquivo único `/etc/pam.conf` e a coleção de arquivos presentes no diretório `/etc/pam.d`. Essencialmente, se você estiver usando o arquivo `/etc/pam.conf`, cada entrada neste arquivo contém um campo guia de tipos de serviços que especifica a aplicação PAM a qual entrada pertence. Se você utilizar `/etc/pam.d`, você irá encontrar uma coleção de arquivos neste diretório cujos nomes refletem as aplicações PAM. Conseqüentemente, o campo tipo de serviço deixa de existir para cada um destes arquivos. Apresenta-se as opções de configuração dos arquivos presentes no diretório `/etc/pam.d`, e este será o método adotado pelo trabalho apresentado por esta monografia.

O conteúdo do diretório `/etc/pam.d` é ilustrado pela Tabela 2.1.

```
# ls /etc/pam.d
chfn              linuxconf-pair   ppp      su
chsh              login            rexec    vlock
ftp               mcserv           rlogin   xdm
imap              other            rsh      xlock
linuxconf         passwd           samba
```

Tabela 2.1: Conteúdo do Diretório `/etc/pam.d`

Esta é normalmente a lista de aplicações PAM que estão embutidas nas distribuições do Red Hat 6.0 ou similares. Cada arquivo aí listado possui uma aplicação PAM a ele associado. Em todos estes arquivos de configuração, as linhas que começam com # são linhas de comentários e são ignoradas pelo PAM.

Cada um dos arquivos de configuração do PAM segue um padrão de entrada, a Tabela 2.2 ilustra o conteúdo do arquivo de configuração `/etc/pam.d/passwd`.

```
auth       required   /lib/security/pam_pwdb.so
account    required   /lib/security/pam_pwdb.so
password   required   /lib/security/pam_cracklib.so  retry=3
password   required   /lib/security/pam_pwdb.so  use_authok
```

Tabela 2.2: Configuração do `passwd`

Atualmente temos quatro campos: `module-type`, `control-flag`, `module-path` e por fim `arguments`. O campo `module-type` especifica o tipo de módulo PAM. Atualmente existem quatro tipos de módulos, `auth, account, session`, e `password`. Eles estão descritos na Tabela 2.3.

Module Type	**Descrição**
`auth`	O módulo `auth` adquire da aplicação (*prompt*) do usuário dados para realizar a identificação, assim como uma senha.
`account`	O módulo `account` verifica os vários aspectos da conta do usuário, como limite de acesso em um período de tempo particular, ou de localização particular. Pode ser usado para limitar o acesso ao sistema baseado no sistema de recurso.
`session`	O módulo `session` é usado para provêr funções antes e depois de estabelecer a sessão. Este contém um conjunto do ambiente, como *logando* etc.
`password`	O módulo `password` é normalmente empilhado no módulo `auth`. Ele é responsável pela atualização do *token* de autenticação do usuário, muita das vezes uma senha.

Tabela 2.3: Tipos de Módulos PAM

Outro campo presente nos arquivos de configuração é o `control-flag`, este especifica a ação a ser tomada dependendo do resultado do módulo PAM. Mais de um módulo PAM pode ser especificado para uma dada aplicação (isto é chamado empilhamento *stacking*). O `control-flag` também determina a relativa importância dos módulos na pilha. Os quatro valores possíveis para este campo são: `required, requisite, optional` e `sufficient`. Estes estão resumidos na Tabela 2.4.

Control Flag	Descrição
`required`	Este módulo deve retornar sucesso para o serviço ser garantido. Se este módulo é um em uma série de uma pilha de módulos, todos os outros módulos ainda são executados. A aplicação não será informada sobre qual ou quais módulos falharam.
`requisite`	Tal como `required`, exceto que a falha aqui, termina a execução para todos os módulos e retorna o *status* da falha para a aplicação.
`optional`	Como o nome sugere, este módulo não é obrigatório. Se ele for o único módulo, entretanto, este retorna um *status* para a aplicação podendo causar uma falha.
`sufficient`	Se este módulo obtiver sucesso, todos os módulos que ficaram na pilha são ignorados e o sucesso é retornado para a aplicação.

Tabela 2.4: *Flag* de Controle PAM

O campo `module-path` indica a localização absoluta dos módulos PAM. Geralmente estes módulos se encontram em `/lib/security`.

O campo `arguments` é usado para especificar *flags* ou opções que são passadas para os módulos. Especificar os argumentos é opcional. Certamente há argumentos gerais que estão disponíveis para a maioria dos módulos listados na Tabela 2.5. Outros argumentos estão disponíveis de acordo com o módulo base [MM00a].

Em resumo, cada arquivo presente no diretório `/etc/pam.d` é associado a um serviço ou aplicação, isto após este arquivo ser nomeado e contiver uma lista com os campos de informação preenchidos, cada um com o `module type`, `control-flag`, nome do módulo, localização e argumentos opcionais. A Tablela 2.2 ilustra o conteúdo do arquivo de configuração `/etc/pam.d/passwd`. Note que há duas entradas para o tipo `password`. Este é um exemplo de entrada na pilha.

2.3 Módulos PAM Disponíveis

A Tabela 2.6 mostra uma lista com alguns módulos PAM disponíveis. Alguns já estão embutidos nas distribuições da Red Hat (ou em outras), enquanto outros precisam ser baixados da rede, podendo encontrá-los no site da RedHat em `http://www.redhat.com/`. Se o seu sistema já tiver suporte a estes módu-

Arguments	Descrição
debug	Gera uma saída adicional para o serviço `syslog`. A maioria dos módulos PAM suporta este argumento. Sua exata definição depende do módulo que este argumento é aplicado.
no_warn	Não retorna mensagem de erro para a aplicação.
use_first_pass	Este módulo usará a senha do módulo anterior. Se ela falhar o módulo não se esforçará para obter outra entrada do usuário. Este argumento tem utilidade para os módulos de `auth` e `password`.
try_first_pass	Assim como o de cima, exceto que, se a senha falhar, será requisitada uma nova entrada do usuário. Este argumento tem utilidade para os módulos `auth` e `password`.

Tabela 2.5: Argumentos Padrão

los, estes serão encontrados em `/lib/security` ou `/usr/lib/security`. Se você fizer o *dowload*, garanta que estes ficarão no diretório correto. Em [Vei02] encontra-se mais informações sobre estes módulos.

Módulos			
pam_acess	pam_console	pam_cracklib	pam_deny
pam_env	pam_filter	pam_ftp	pam_group
pam_if	pam_lastlog	pam_limits	pam_listfile
pam_mail	pam_nologin	pam_opie	pam_permit
pam_pwdb	pam_pwdfile	pam_radius	pam_rhosts_auth
pam_rootok	pam_securetty	pam_shells	pam_stress
pam_tally	pam_time	pam_tcpd	pam_unix − new
pam_warn	pam_wheel	pam_xauth	

Tabela 2.6: Módulos PAM disponíveis

2.4 Desenvolvimento de Módulos PAM

Linux-PAM (*Pluggable Authentication Modules for Linux*) é uma coleção de programas que permitem ao administrador do sistema local escolher quais aplicações irão utilizar quais módulos.

Um módulo Linux-PAM é um único arquivo binário executável que pode ser carregado por aplicações que mantém um arquivo de configuração presente no diretório `/etc/pam.d`.

Para construir um módulo PAM você deve inserir em seu código fonte a seguinte biblioteca : `#include <security/pam_modules.h>` e compilar da seguinte forma:

gcc -fPIC -c pam_module-name.c
ld -x –shared -o pam_module-name.so pam_module-name.o

2.4.1 Apontando Dados

Sinopse:

```
extern int pam_set_data(pam_handle_t * pamh,
                        const char *module_pam_name,
                        void *data,
                        void (*cleanup)(pam_handle_t *pamh,
                                        void *data, int error_status));
```

Normalmente um módulo chama a função `pam_set_data()` para registrar algum dado exclusivamente para um `module_pam_name`. Este dado está também disponível para outros módulos mas não para uma aplicação.

A função `cleanup()` está associada a `data` e se este dado não for nulo, esta função é chamada quando este dado é sobreposto ou obedeçendo uma chamada `pam_end()`.

O argumento `error_status` é usado para indicar ao módulo a maneira de agir, ao adiquirir este dado. Este argumento `error_status` deve ser associado logicamente a dois valores:

- `PAM_DATA_REPLACE:` Quando o dado é reposto (através de uma segunda chamada ao `pam_set_data()`, esta máscara é usada. Se não, a chamada é suposta de `pam_end()`.

- `PAM_DATA_SILENT:` Demostra que o processo escolherá executar a função `cleanup()`.

2.4.2 Pegando Dados

Sinopse:

```
extern int pam_get_data(const pam_handle_t *pamh,
                        const char *module_data_name,
                        const void **data);
```

Esta função juntamente da apresentada anteriormente fornecem um método para associar um dado de um módulo específico. Uma chamada bem sucedida a função `pam_get_data` irá resultar em um ponteiro `*data`, associado ao dado do `module_data_name`. Se a entrada for nula, então a chamada a esta função retorna `PAM_NO_MODULE_DATA`.

2.4.3 Apontando Itens

Sinopse:

```
extern int pam_set_item(pam_handle_t *pamh,
                        int item_type,
                        const void *item);
```

Esta função é usada para apontar (*set*) ou reapontar (*reset*) o valor de um dos `item_types`. O módulo pode apontar um dos dois valores para `item_types`:

- `PAM_AUTHTOK`: Um sinal de autenticação freqüente como *password*, deveria ser ignorado por todas as funções do módulo, exceto para as funções `pam_sm_authenticate()` e `pam_sm_chauthtok()`. Na função anterior `pam_sm_authenticate()` ela é usada para passar o sinal mais recente de autenticação de um módulo empilhado para outro. Na segunda função `pam_sm_chauthtok()` o sinal é usado para outro propósito. Esta contém o sinal atual de autenticação.

- `PAM_OLDAUTHTOK`: Tem o valor velho do sinal de autenticação, o valor anterior à atualização. Este sinal deveria ser ignorado por todas funções dos módulos exceto por `pam_sm_chauthtok()`.

Ambos itens são zerados (*reset*), antes de retornarem para a aplicação.

2.4.4 Pegando Itens

Sinopse:

```
extern int pam_get_item(const pam_handle_t *pamh,
                        int item_type,
                        const void **item);
```

Esta função é usado para se obter um valor específico de um `item_type`. Normalmente se o módulo quer obter o nome do usuário não se usa esta função, mas sim uma chamada a função `pam_get_user()`.

2.4.5 Obtendo o Nome de Usuário

Sinopse:

```
extern int pam_get_user(pam_handlea_t *pamh,
                        const char **user,
                        const char *prompt);
```

Esta função pode retornar os seguintes valores:

- `PAM_SUCCESS`: Nome de usuário recebido.
- `PAM_CONV_AGAIN`: A comunicação não foi completada e o controle é devolvido para a aplicação para que este tente obter novamente o nome de usuário.
- `PAM_CONV_ERR`: Devolve um erro ao tentar obter o nome de usuário.

2.4.6 Gerenciamento de Autenticação

Apresenta-se as funções para o gerenciamento de autenticação e seus *flags*:

- `PAM_EXTERN int pam_sm_authenticate(pam_handle_t *pamh, int flags, int argc, const char **argv);`
  Esta função executa a tarefa de autenticar um usuário. Os argumentos *flags* podem ser `PAM_SILENT` e opcionalmente assume os seguintes valores:
  - `PAM_DISALLOW_NULL_AUTHTOK:` Retorna `PAM_AUTH_ERR` se o banco de dados de autenticação de *tokens* para este mecanismo de autenticação receber uma entrada `NULL` para o usuário. Além de retornar valores `PAM_SUCCESS` esta função também pode retornar valores:
  - `PAM_AUTH_ERR:` O usuário não foi autenticado.
  - `PAM_CRED_INSUFFICIENT:` Por alguma razão a aplicação não tem credenciais suficientes para autenticar o usuário.
  - `PAM_AUTHINFO_UNAVAIL:` Os módulos não estão habilitados para acessar informações de autenticação. Isto pode ser devido uma falha na rede ou no *hardware*.

  - `PAM_USER_UNKNOWN:` O *username* fornecido não é conhecido do serviço de autenticação.
  - `PAM_MAXTRIES:` Um ou mais módulos de autenticação ultrapassou o limite de tentativas para autenticar um usuário.

- `PAM_EXTERN int pam_sm_setcred(pam_handle_t *pamh, int flags, int argc, const char **argv);`
  Esta função executa a tarefa de modificar as credencias de usuários respeitando o esquema de de autorização correspondente. Geralmente, um módulo de autenticação pode ter acesso a mais informações sobre um usuário que seu sinal (*token*) de autenticação. Esta função é usada para fazer com que esta informação fique disponível para a aplicação. Esta função deve ser chamada após o usuário ter sido autenticado. Os *flags* são:

  - `PAM_ESTABLISH_CRED:` Aponta as credencias para o serviço de autenticação.
  - `PAM_DELET_CRED:` Apaga as credencias associadas com o serviço de autenticação.
  - `PAM_REINITIALIZE_CRED:` Reinicia as credenciais do usuário.
  - `PAM_REFRESH_CRED:` Prolonga a validade das credencias do usuário.

Capítulo 3

Comunicação Segura

3.1 Criptografia

O crescimento contínuo da Internet e das redes privadas de computadores traz consigo o desafio de garantir a segurança das informações que trafegam por essas redes. Assim a necessidade de prover a segurança dessas informações torna-se cada vez mais importante.

Dentre as maneiras de garantir segurança pode-se citar a criptografia. Neste trabalho a criptografia será utilizada para garantir uma maior segurança quando as senhas estiverem trafegando das estações para o servidor de autenticação.

Esta seção tem por objetivo conceituar e apresentar um breve histórico sobre criptografia, e apresentar alguns modelos de implentação.

3.1.1 Conceitos e Históricos

A palavra criptografia é definida pelo dicionário Aurélio como "a arte de escrever em cifra ou em código", significado este que remete ao grego *cryptos* que significa secreto, oculto. Em [Fer95] encontra-se o modelo de criptosistema assim descrito: "dada uma mensagem e uma chave de codificação como entrada, o método de codificação produz como resultado uma mensagem codificada, a qual pode ser armazenada num meio qualquer ou enviada a um destinatário; para decodificar esta mensagem utiliza-se o método de decodificação passando como entradas a mensagem codificada e a chave de decodificação obtendo-se a mensagem original", como ilustrado na Figura 3.1.

Uma complementação ao conceito de criptografia nos remete a sua finalidade,

Figura 3.1: Criptografar e Descriptografar

como visto em [Col00] "a criptografia estuda os métodos para codificar uma mensagem de modo que só seu destinatário legítmo consiga interpretá-la, é a arte dos códigos secretos", considere como destinatário legítimo o servidor responsável pela autenticação dos usuários.

Paralelamente à criptografia, mas trilhando o caminho inverso, objetivando decifrar os códigos secretos está a criptoanálise. Nesse ponto vale salientar a diferença entre os termos decodificar e decifrar para evitar quaisquer ambigüidade quanto aos seus significados. Conforme [Col00], decodificar é o que um usuário legítimo do código faz quando recebe um texto codificado e deseja vê-lo. E decifrar significa ler um texto codificado sem ser o usuário legítimo de tal texto, portanto para decifrar um texto é necessário *quebrar* o código.

Como a criptografia é uma técnica de milhares de anos, sua longevidade fez surgir uma classificação da mesma em *Criptografia Tradicional* e *Criptografia Moderna ou Computacional*.

- **A Criptografia Tradicional:** Conforme visto em [Fer95] antes da era computacional, a criptografia baseava-se na substituição de um caracter por outro ou na troca de posição dos caracteres no texto, sendo por isso denominado criptografia orientada a caracter. Para aumentar a segurança, alguns métodos utilizavam tanto substituição quanto a troca de posição dos caracteres e de preferência várias vezes. Todos estes métodos são simétricos.

- **A Criptografia Moderna ou Computacional:** Com o grande avanço da tecnologia computacional, procedeu-se a substituição dos métodos criptográficos tradicionais por métodos criptográficos computacionais, onde as operações são implementadas por um computador ou por um circuito integrado especial, tendo como conseqüência a aceleração dos processos de codificação e decodificação [Oli02].

3.1.2 O propósito da Criptografia

A criptografia é utilizada para vários propósitos, todos objetivando guardar em sigilo suas informações, sejam elas quais forem. Podendo ser dados pessoais, transações bancárias, informações guardadas em cartões inteligentes, transmissões via satélite etc .

Como a presente monografia trata a área de comunicação em redes de computadores, foca-se o uso de algoritmos criptográficos para esta área. Segundo [MM00b] devemos enfocar algoritmos para grarantir a confidencialidade, autenticação e integridade, para aplicações de redes de computadores:

- **Confidencialidade:** Permite que a informação seja avaliada apenas para aqueles participantes de uma comunicação privada. Historicamente, confidencialidade era o uso predominante da criptografia.
- **Autenticação:** É o processo de verificação da identidade de um indivíduo em particular. Autenticar um usuário através de criptografia significa validar a identidade deste usuário.
- **Integridade:** É usado para garantir a integridade dos dados, garantir que estes não sofreram nenhuma modificação ou foram corrompidos.

3.1.3 Exemplos de Métodos Criptográficos

Esta seção apresentará alguns métodos criptográficos existentes, tendo o intuito de mostrar sucintamente algumas idéias utilizadas para criptografar dados.

Na criptografia convencional, também chamada de criptografia de chave secreta ou criptografia de chave simétrica, uma única chave é usada para criptografar e para descriptografar. O algoritmo *Data Encryption Standard* (DES) [Sch96] é um exemplo de criptografia convencional que foi utilizada pelo Governo Federal Americano [Netb]. O processo de criptografia convencional pode ser visto na Figura 3.2. Um exemplo bem simples de criptografia de chave simples é o de substituir alguns caracteres por outros, ou deslocar o texto em um certo número de caracteres.

Um exemplo histórico é o que foi utilizado pelo imperador romano Júlio Cesar, o método consiste em susbtituir os caracteres por outros caracteres que estão avançados em três posições no alfabeto [Netb]. As letras que estão no texto criptografados seriam na realidade as letras:

$$D = A, E = B, F = C, G = D, \text{ e assim por diante,}$$

então as palavras "casa", "Paulo" e "criptografia" criptografadas por este método seriam representadas da seguinte maneira:

$$casa = fdvd, \quad Paulo = Sdxnr, \; criptografia = fukswrjudikd$$

Para descriptografar basta usar a mesma chave, porém deslocando as letras no alfabeto em três caracteres para a esquerda.

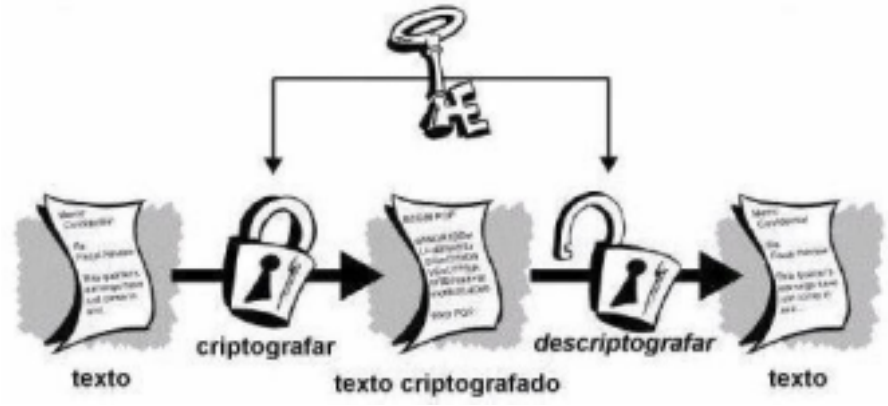

Figura 3.2: Criptografia Convencional ou Criptografia de Chave Simétrica

De acordo com [Netb] os algoritmos de criptografia de chave simétrica tem uma grande vantagem, eles são muito rápidos. Neste método a chave deve ser enviada para o destinatário para que este possa descriptografar o texto, o problema é que se este texto for interceptado, juntamente dele estará a chave para descriptografar o texto.

O problema de distribuição de chaves é resolvido pela criptografia de chave pública, conceito introduzido por Whitfild Diffie e Martin Hellman em 1975 [Sch96]. Criptografia de chave pública é um esquema assimétrico que usa um par de chaves para criptografar: a *chave pública*, criptografa os dados e a correspondente *chave privada* ou *chave secreta*, que descriptografa os dados. Divulga-se a chave pública para o mundo, enquanto a chave privada deve ser secreta. Cada um que tiver a cópia de sua chave pública pode criptografar informações que somente você poderá ler. É computacionalmente impraticável a dedução da chave privada a partir

da chave pública [Sch96]. Cada um que tiver a chave pública pode criptografar o dados, porém não pode descriptografá-los. Somente as pessoas que tiverem a chave privada poderão ler a mensagem, como ilustra a Figura 3.3.

Figura 3.3: Criptografia de Chave Pública

O benefício primário do método de criptografia de chave pública e que ele permite que pessoas que não tenham um método de criptografia possam usar a chave pública de um dos envolvidos na troca dos dados, garantindo que somente o destinatário terá acesso às informações. Com este método o remetente e o destinatário não precisam dividir um canal seguro para poder transmitir as chaves, pois somente a chave pública será transmitida pela rede.

Apresenta-se alguns exemplos de algoritmos criptograficos, primeiramente apresenta-se os algoritmos criptográficos simétricos e posteriormente os algoritmos criptográficos assimétricos. Entre os algoritmos simétricos, tem-se:

- **Blowfish:** Tal algoritmo de criptografia em bloco foi inventado por Bruce Schneier [Sch96] e permite a utilização de chaves de até 448 *bits*, sendo otimizado para ser executado em máquina de 32 ou 64 *bits*.

- **IDEA:** O algoritmo de criptografia de dados internacional - IDEA foi desenvolvido na Suíça por James Massey e Xuenjia Lai e publicado em 1990 [Sch96]. Ele é um algoritmo de blocos com tamanho de 64 *bits* e utiliza chaves de 128 *bits*. Acredita-se que ele seja um algoritmo bastante poderoso, visto que ainda não existe nenhum método de ataque efetivo contra o

mesmo e também por ele ter resistido bem contra métodos aplicados com êxito sobre outros algoritmos [Fer95].

- **DES:** O DES(*Data Encrytion Standard*) [Sch96] é o exemplo mais difundido de algoritmo criptográfico de chave única. Ele foi desenvolvido pela IBM e adotado pelos Estados Unidos em 1977. O DES é um algoritmo de bloco e trabalha dividindo o texto em blocos de 8 caracteres, cifrando cada bloco com uma chave de 56 *bits* (mais 8 *bits* de paridade, totalizando uma chave de 64 *bits*. Esse método é passível de ser quebrado usando o método da força bruta, bastando para tal testar as 2^{56} chaves possíveis [Sch96].

- **Triple-DES:** Este algoritmo [Sch96] é um método para tornar o DES mais seguro e para atingir tal objetivo aplica-se o algoritmo do DES três vezes com duas chaves diferentes: inicialmete cifra-se o texto com a chave *C1*, depois com a chave *C2* e finalmente com a chave *C1* novamente. Para quebrar tal método com o método da força bruta é necessário testar as 2^{112} chaves possíveis.

Descrito alguns algoritmos simétricos, descreve-se alguns métodos de cripotografia assimétrica ou de chave pública.

- **RSA:** Este método de criptografia com chave pública ou assimétrico foi proposto em 1977 por Rivest, Shamir e Adleman [Sch96] e seu nome deriva das iniciais dos sobrenomes de seus inventores. Este método basea-se em uma chave pública composta por *n* e *e*, onde *n* é o produto entre dois números primos e *e* é um número inteiro positivo que seja inversível módulo $\phi(n)$. Já a chave privada é composta por *n* e *d*, onde *n* como já dito é o produto de dois números primos e *d* é o inverso de *e* em $\phi(n)$.

- **Rabin:** Este método [Sch96] baseia-se na dificuldade de se extrair a raiz quadrada em aritmética modular de um número composto. Para aplicá-lo escolhe-se dois números primos, *p* e *q*, sendo que ambos devem ser congruentes a 3 módulo 4 e tais primos são a chave privada. Já a chave pública é calculada pelo produto de *p* e *q*. Pode-se encontrar mais detalhes em [Fer95] e [Sch96], sobre este algoritmo.

- **ElGamal:** Criado por Taher ElGamal [Sch96] é um sistema criptográfico de chave pública que pode ser usado tanto para cifrar mensagens quanto para assinatura digital. Tem sua segurança baseada na dificuldade de calcular logaritmos discretos e aritmética modular.

Além destes métodos acima citados, podem ser encontrados outros na literatura, tais como: DSA, LUC, DSS entre outros. Recomenda-se aos interessados verificar em [Fer95] e [Sch96] os quais indicam uma série de algoritmos de criptografia de chave pública com uma sucinta descrição sobre os mesmos.

3.1.4 Senhas no Linux

As senhas no Linux não são criptografadas elas são codificadas (*hashed*) [MM00b]. O algoritmo utilizado pelo Linux para codificar as senhas é um algoritmo de *hash*, veja o esquema na Figura 3.4, conhecido como função `crypt (3)`. Este algoritmo implementa uma modificação do algortimo DES, que aceita um máximo de 64 *bits* de entrada (8 caracteres) e produz um valor *hash* de 104 *bits* (13 caracteres, veja o manual do Linux `[man 3 crypt]`). Esta é a função usada como padrão para função de codificação das senhas no Linux. Cada senha codificada dos usuários é armazenada em `/etc/shadow` ou em `/etc/passwd`. Algoritmos *hash* podem ser encontrados em [Sch96].

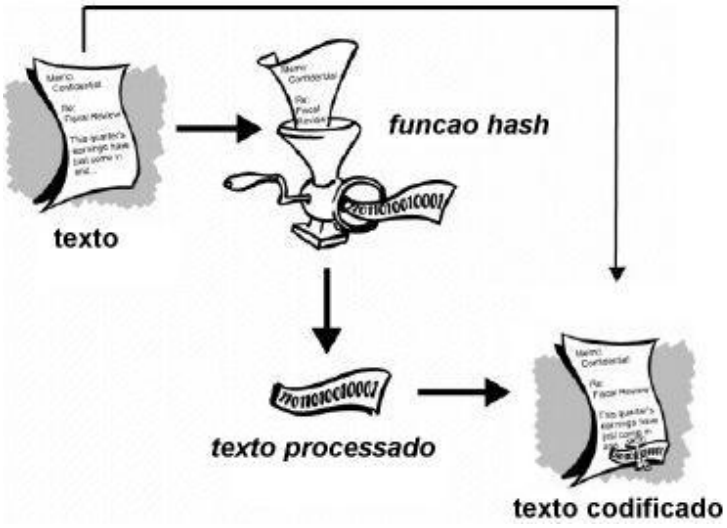

Figura 3.4: Codificação Utilizando uma Função Hash

Porém na maioria das vezes a senha codificada é freqüentemente chamada de senha criptografada, mas o termo criptografada implica que o dado pode ser descriptografado. Segundo [MM00b] pode-se descodificar uma senha codificada por

uma função *hash*, porém isto é computacionalmente difícil e não é feito. Ao invés disso, quando um usuário entra com sua senha, esta senha é codificada pelo algoritmo `crypt (3)` e o resultado é comparado com a senha que está armazenada em `/etc/shadow`.

3.2 Sockets

Segundo [MCCI99] um computador é identificado na Internet por um número, o endereço IP. Já um processo (um programa), que está em execução em um computador é identificado por um número de porta. Programas utilizam números de porta para aplicativos cliente/servidor. Como os endreços IP são únicos e, em um determinado computador, os números de porta também são únicos, o número de porta combinado com um endereço IP diferencia totalmente um aplicativo em execução na Internet. A combinação do número de porta e do endereço IP resulta em um *socket* e fornece um ponto final no caminho da comunicação.

Um socket é um mecanismo de comunicação que permite sistemas cliente/servidor serem desenvolvidos localmente em uma simples máquina, ou através da rede. Funções UNIX, como `printing`, e utilidades de rede, como `rlogin` e `ftp`, normalmente usam sockets para se comunicarem [SM99a].

3.2.1 Compreendendo uma Conexão de Socket

Quando dois aplicativos querem se comunicar, eles precisam encontrar um ao outro na rede (Internet). Geralmente é responsabilidade do cliente encontrar o servidor [MCCI99]. Por exemplo, para telefonar para pedir que entreguem comida chinesa na sua casa, você precisa saber o número do telefone do restaurante de comida chinesa. O mesmo também ocorre com um aplicativo cliente. Um cliente se conecta com um servidor iniciando uma conexão de socket. O aplicativo cliente faz isso enviando uma mensagem (seu número de socket) ao socket do servidor.

Por outro lado, o restaurante de comida chinesa simplesmente espera que os clientes façam contato. Isso também ocorre com os aplicativos servidores. O servidor não precisa saber inicialmente onde está o cliente, porque está esperando que o cliente faça contato. Mas, depois de você encomendar seu frango xadrez, o restaurante vai requisitar seu endereço e número de telefone. Agora ambos os lados já têm um modo de começar a comunicação. Se alguma coisa der errado, por exemplo, se o entregador se perdeu ou se você quiser mudar o seu pedido, serão iniciadas as ações corretivas. Foi estabelecido um caminho de comunição

confiável. A primeira mensagem que o aplicativo cliente envia ao servidor contém o *socket* do cliente.

Por sua vez, o servidor cria um *socket* que utilizará para se comunicar com o cliente, e envia esse endereço de *socket* ao cliente, na sua primeira mensagem de retorno.

3.2.2 As Principais Chamadas Sockets

Segundo [Rib00] as chamadas de socket podem ser separadas em dois grupos: **chamadas primárias** que fornecem acesso a funcionalidades subjacentes e **rotinas de utilidade** que ajudam o programador. Esta secção descreve as chamadas que fornecem as funcionalidades primárias que os clientes e os servidores necessitam.

- **A Chamada Socket:** Uma aplicação chama o `socket()` para criar um novo socket que possa ser usado para comunicação em rede. A chamada retorna um descritor para o novo socket. Argumentos para a chamada especifica o protocolo da família que a aplicação irá usar (por exemplo PF_INET para o TCP/IP) e o protocolo ou tipo de serviço que precisa (isto é fluxo ou datagramas). Para um socket que usa um protocolo da família da internet, o protocolo ou tipo de argumento do serviço determina se o socket irá usar o TCP ou UDP.

- **A Chamada Connect:** Após criar um socket, um cliente chama então a função `connect()` para estabelecer uma conexão ativa para um servidor remoto. Um argumento do `connect()` permite ao cliente especificar o *endpoint* remoto, que inclui o endereço IP remoto da máquina e o número da porta do protocolo. Uma vez feita uma conexão, o cliente pode transferir dados através desta.

- **A Chamada Write:** Ambos os clientes e os servidores usam `write()` para mandar informação através da conexão TCP. Os clientes usualmente utilizam o `write()` para mandar pedidos , enquanto os servidores usam-na para mandar as respostas. A chamada ao `write()` requer três argumentos. A aplicação passa o descritor do soket para onde a informação deve de ir, o endereço da informação a ser mandada e o tamanho da informação. Usualmente, `write()` copia a informação que vai mandar para *buffers* no *kernel* do sistema operacional, e permite a aplicação continuar a execução enquanto transmite a informação para a rede. Se o *buffer* do sistema tornar-se cheio

a chamada `write()` pode bloquear temporariamente até que o TCP possa mandar a informação através da rede e haja espaço no *buffer* para nova informação.

- **A Chamada Read:** Ambos os clientes e os servidores usam o `read()` para receber dados da conexão TCP. Usualmente, após uma conexão ter sido estabelecida, o servidor usa o `read()` para receber o pedido que o cliente manda através da chamada `write()`. Após mandar o pedido o cliente usa o `read()` para receber uma resposta do servidor.

  Para ler de uma ligação, uma aplicação chama o `read()` com três argumentos. O primeiro especifica o descritor do socket a ser usado, o segundo especifica o endereço de um *buffer* e o terceiro especifica o tamanho deste *buffer*. O `read()` extrai dados em bytes que vão chegando ao socket, e copia-os para a área do *buffer* do utilizador. Se nenhum dos dados chegarem, a chamada ao `read()` bloqueia até chegar uma informação. Se os dados extrapolam o tamanho do *buffer*, o `read()` apenas extrai dados suficientes para encher o *buffer*. Se chegou menos dados do que se encaixam no *buffer*, o `read()` extrai todos os dados e retorna o número de bytes que encontrou.

  Clientes e servidores podem usar ainda o `read()` para receber mensagens do socket que usa UDP. Como o caso de conexão orientada, quem chama fornece três argumentos: que identificam o descritor do socket, o endereço de um *buffer* em que a informação será colocada e o tamanho deste *buffer*. Cada chamada ao `read()` extrai uma mensagem UDP (isto é um datagrama do utilizador). Se o *buffer* não pode guardar a mensagem toda, `read()` preenche o *buffer* e deixa fora o restante.

- **A Chamada Close:** Quando um cliente ou um servidor termina o uso de um socket, este chama o `close()` para desalocar este socket. Se vários processos partilham um mesmo socket, o `close()` termina imediatamente a conexão.

- **A Chamada Bind:** Quando um socket é criado, não têm qualquer noção do endereço *endpoint* (nem o local nem o endereço remoto são atribuídos). Uma aplicação chama o `bind()` para especificar o endereço *endpoint* local para um socket. A chamada toma argumentos que especifica o descritor do socket e um endereço *endpoint*. Para protocolos TCP/IP, o endereço *endpoint* usa a estrutura `sockaddr_in`, que inclui ambos um endereço IP

e um número da porta do protocolo. Primeiramente, os servidores usam o `bind()` para especificar a porta onde será esperada as conexões.

- **A Chamada Listen:** Quando um socket é criado, este socket nem é **ativo** (isto é está pronto para ser usado pelo cliente) nem **passivo** (isto é está pronto para ser usado pelo servidor) até a aplicação dar o próximo passo. Servidores orientados à conexão chamam o `listen()` para colocar um socket em modo passivo e deixa-lo pronto para receber conexões que vão chegar.

  Muitos servidores consistem num *loop* infinto que espera por uma conexão, quando esta chega o servidor trata-a, e depois volta para esperar a próxima conexão. Pode acontecer que uma nova conexão chegue durante o tempo em que o servidor está ocupado com um pedido existente, mesmo que este dure apenas alguns milesegundos. Para assegurar que nenhum pedido é perdido, um servidor deve passar para o `listen()` um argumento que diz ao sistema para enfileirar a conexão requerida, (coloca-la em um fila) para um determinado socket. Assim, um argumento à chamada `listen()` especifica um socket para ser colocado em modo passivo, enquanto a outra especifica o tamanho da fila que vai ser usada por esse socket.

- **A Chamada Accept:** Após o servidor chamar o `socket()` para criação de um socket, o `bind()` para especificar o endereço *endpoint* local, então chama o `listen()` para o colocar no modo passivo, o servidor chama o `accept()` para extrair o próximo pedido de conexão.

  Então `accept()` cria um novo socket para cada novo pedido de conexão, e retorna o descritor do novo socket para quem o chamou. O servidor usa o novo socket para a nova conexão e usa o socket original para aceitar pedidos de conexões adicionais.

  Uma vez aceita a conexão, o servidor pode transferir informação no novo socket. Após finalizar o uso do novo socket, o servidor fecha-o.

3.2.3 Uma Típica Conversação Cliente/Servidor

A idéia básica por trás de *sockets* (assim como toda base-TCP de serviços cliente/servidor) é que o servidor senta e espera por conexões através da rede na porta em questão. Quando um cliente conecta através desta porta, o servidor aceita a conexão e ele conversa com o cliente usando algum protocolo, seja lá qual for, por eles estabelecidos (como HTTP, NNTP, SMTP etc.).

Inicialmente, o servidor usa a chamada `socket()` para criar um socket, e a chamada `bind()` para designar o socket para uma porta particular no *host*. O servidor então usa as rotinas `listen()` e `accept()` para estabelecer a comunicação naquela porta.

Por sua vez, o cliente também usa a chamada `socket()` para criar um socket, e então a chamada `connect()` para iniciar uma conexão associada a este socket a um *host* remoto específico e porta.

O servidor usa a chamada `accept()` para captar a conexão chegada e iniciar uma comunicação com o cliente. Agora o cliente e o servidor podem cada um usar as chamadas `read()` e `write()` para se comunicarem, antes da transação terminar.

Finalmente, um dos dois o cliente ou o servidor usa a rotina `close()` para terminar a conexão. A Figura 3.5 mostra o fluxo da transação dos sockets.

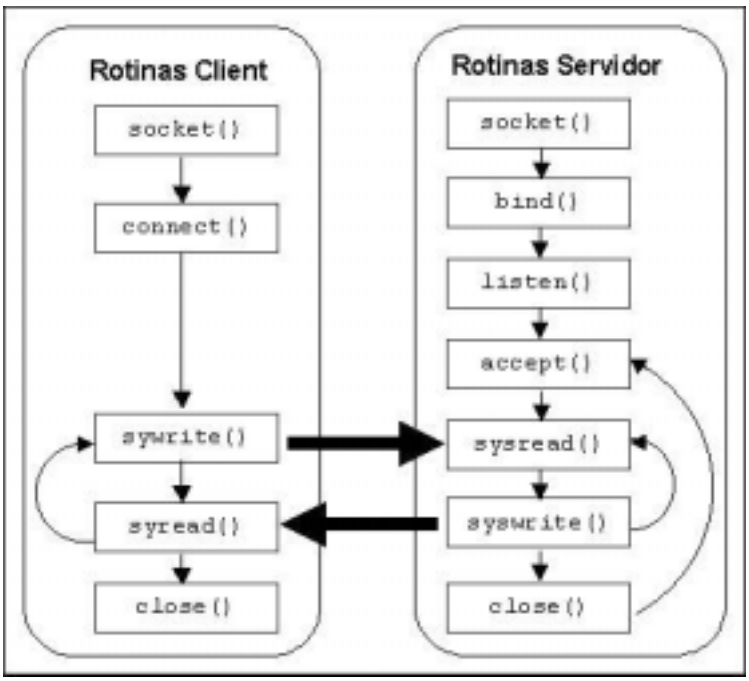

Figura 3.5: Fluxo da Transação de Sockets

3.3 SSL (*Secure Sockets Layer*)

O SSL (*Secure Sockets Layer*) é um protocolo de segurança criado pela *Netscape Corporation* para prover autenticação e cifragem sobre redes TCP/IP, inclusive a internet. O SSL foi projetado para rodar sobre protocolos de transporte confiáveis [Neta]. Outro protocolo de transmissão confiável de dados é o TLS (*Transport Layer Security*) [SB02]. No protocolo SSL, clientes e servidores podem se autenticar e então trocar dados cifrados entre si. As principais características do SSL são:

- **Segurança em conexões cliente/servidor:** O SSL garante o sigilo dos dados trocados entre as partes envolvidas na conexão através do uso de criptografia simétrica. A fim de evitar que as mensagens, mesmo decifradas, sejam modificadas e com isso um ataque de escuta ativa seja possível, o SSL adiciona à todas as mensagens um MAC (*Message Authentication Code*). Calculado a partir de funçõeses de *hash* seguras, o MAC garante a integridade das mensagens trocadas. Além de sigilo e integridade, o SSL ainda provê a autenticação das partes envolvidas a fim de garantir e verificar a identidade das mesmas. Neste processo, o SSL utiliza criptografia assimétrica e certificados digitais.

- **Independência de protocolo:** O SSL roda sobre qualquer protocolo de transporte confiável. Porém, a maioria das implementações são feitas para redes TCP/IP.

- **Interoperabilidade:** Dado a sua especificação bem detalhada e o uso de algoritmos criptográficos conhecidos, diferentes implementações do protocolo tem a garantia de interagir entre si.

- **Extensibilidade:** Dado a necessidade, permitir que novos parâmetros e métodos de criptografia (assimétrica ou simétrica) sejam incorporados ao protocolo, sem que seja necessária a criação de um novo protocolo ou a implementação inteira de uma nova biblioteca.

- **Eficiência:** Devido a demanda por recursos computacionais que este tipo de operação requer, o protocolo dispõe da opção de armazenamento em cache de informações referentes a sessão, diminuindo desta forma o esforço computacional em sucessivas conexões.

3.3.1 Camadas

O protocolo SSL é dividido em duas camadas [Ope]. A de mais baixo nível, que interage com o protocolo de transporte, e a camada *Record*. Esta camada é responsável por encapsular os dados das camadas superiores em pacotes compactados e cifrados e repassá-los para a camada de transporte. Entre as camadas superiores está a outra camada do SSL, a camada de *Handshake*. Esta camada permite que a aplicação servidora e a aplicação cliente autentiquem-se e negociem os algoritmos de cifragem e as chaves criptográficas antes que o protocolo de aplicação receba ou envie seu primeiro *byte*.

A camada *Record* recebe dados não interpretados de camadas superiores em forma de bloco de dados cujo tamanho é variado. Estes blocos são encapsulados em registros e, dependendo do seu tamanho, devem sofrer uma fragmentação. Os dados contidos neste registro sofrem ainda uma compactação e, em seguida, são cifrados usando os algoritmos e chaves definidos pelo processo de *handshake* [Ope].

A camada de *Handshake* é a responsável pelos processos de troca de chaves, autenticação e estabelecimento de chave de sessão feitas no SSL. Nela, encontram-se os protocolos *Handshake, ChangeCipherSpec* (CCS) e *Alert*. Um conceito importante para o entendimento desta camada é o de sessão no SSL. Uma sessão SSL é composta por um conjunto de dados que são gerados após um processo de *handshake* completo. Uma sessão é dada como sendo dependente do estado. O protocolo *Handshake* é o responsável por manter a consistência dos estados de uma sessão tanto no cliente quanto no servidor. Uma mesma sessão SSL pode incluir várias conexões, ou seja, a partir dos mesmos dados que formam uma sessão pode-se abrir múltiplas conexões SSL. Os dados que formam uma sessão são os seguintes:

- **session ID:** Um valor arbitrário escolhido pelo servidor para identificar esta sessão;
- **peer certificate:** Usado para certificar uma organização. Está no formato X.509 e dentre outras coisas encontra-se dentro dele a chave pública da entidade que está utilizando aquela aplicação;
- **compression method:** Algoritmo usado na compressão dos dados;
- **cipherspec:** Especifica que conjunto de algoritmos de cifragem e de hash serão utilizados;

- **Mastersecret:** Um segredo de 48 *bytes* compartilhado pelo servidor e pelo cliente;
- **IsResumable:** *Flag* utilizada para indicar se a sessão pode ou não ser retomada ao iniciar uma nova conexão.

3.3.2 Comentários Finais

Com o uso de SSL e sockets pode-se implementar aplicações cliente/servidor, garantindo-se a segurança e integridade dos dados que estarão sendo transmitidos entre cliente e servidor.

Capítulo 4

Como é Feita a Autenticação em Linux

Um dos principais requisitos exigidos de um sistema é segurança. Um sistema seguro é aquele que permite que as informações possuam integridade, e que os acessos a essas informações sejam feitos por pessoas autorizadas, permitindo proteger as informações contra acessos indesejáveis.

É importante frisar que não existe um sistema que pode ser completamente seguro [Con]. Tudo que se pode fazer é aumentar a dificuldade de invasões no sistema. É necessário estabelecer uma política de segurança, estabelecendo o quanto de segurança é necessário no sistema e qual a melhor maneira de implementá-la para este nível. Por exemplo, a implementação de uma rede segura pode ser diferente da implementação de um *site* seguro, que pode diferir de proteger apenas um servidor dentro da rede.

Existem vários métodos simples que podem ser verificados para aumentar a segurança de um sistema: quem pode acessar fisicamente uma máquina essencial, segurança na BIOS, gerenciador de inicialização com senha, privilégios para usuários, verificação de portas para Internet, compilação de módulos de segurança no kernel, enfim, uma infinidade de detalhes que podem ser verificados pelo administrador do sistema como tarefa rotineira.

Neste contexto incluí-se o conceito de autenticar. Autenticação é o processo de reconhecimento dos dados que são recebidos, comparando-os com os dados que foram enviados, e verificando se o transmissor que fez a requisição é, na verdade, o transmissor real.

Autenticação utiliza o modelo cliente-servidor. Um cliente faz a requisição

para o servidor, que verifica se o cliente tem permissão para acessar o sistema. Este também verifica quais são estas permissões, ou seja, quais as informações o cliente pode acessar. Após isso, retorna uma resposta à requisição do cliente.

Neste capítulo apresenta-se três soluções que implementam autenticação em Linux: LDAP, que trabalha com serviço de diretório, NIS que trabalha com serviços e compartilhamento de informações de usuários na rede, e a solução Radius e Portslave, que é um protocolo de autenticação e um emulador de hardware que utiliza este protocolo.

4.1 LDAP

LDAP é um protocolo (executado sobre o TCP/IP) cliente-servidor, utilizado para acessar um serviço de Diretório. Ele foi inicialmente usado como uma interface para o X.500 [YHK95], mas também pode ser usado com autonomia e com outros tipos de servidores de Diretório [Con]. Atualmente vem se tornando um padrão, diversos programas já têm suporte a LDAP. Livros de endereços, autenticação, armazenamento de certificados digitais (S/MIME) e de chaves públicas (PGP) são alguns dos exemplos onde o LDAP já é amplamente utilizado.

4.1.1 Serviço de Diretório

Um Diretório é como um banco de dados, mas tende a conter mais informações descritivas, baseadas em atributos e é organizado em forma de árvore, não de tabela. A informação em um Diretório é geralmente mais lida do que é escrita [Con]. Como conseqüência, Diretórios normalmente não são usados para implementar transações complexas, ou esquemas de consultas regulares em bancos de dados, transações estas que são usadas para fazer um grande volume de atualizações complexas.

Diretórios são preparados para dar resposta rápida a um grande volume de consultas ou operações de busca. Eles também podem ter a habilidade de replicar informações extensamente; isto é usado para acrescentar disponibilidade e confiabilidade, enquanto reduzem o tempo de resposta.

Existem várias maneiras diferentes para disponibilizar um serviço de Diretório. Métodos diferentes permitem que diferentes tipos de informações possam ser armazenadas no Diretório, colocando requerimentos diferentes, sobre como aquela informação poderá ser referenciada, requisitada e atualizada, como ela é protegida de acessos não autorizados, etc. Alguns serviços de Diretório são locais,

fornecendo o serviço para um contexto restrito (exemplo: o serviço `finger` em uma máquina isolada). Outros serviços são globais, fornecendo o serviço para um contexto muito maior (por exemplo, a própria Internet).

4.1.2 Acessando a Informação

O LDAP define operações para consultar e atualizar o Diretório. Operações são fornecidas para adição e remoção de uma entrada do Diretório, modificação de uma entrada existente e modificação do nome de uma entrada. A operação LDAP de busca pode abranger a árvore toda (uma busca com escopo *subtree*) ou apenas um ramo, sem descer ou subir para os demais. Além de especificar com filtros quais entradas se deseja encontrar, também é possível especificar quais atributos destas entradas estão sendo procurados. Se os atributos não forem especificados, todos serão retornados.

4.1.3 Funcionamento do LDAP

O serviço de Diretório LDAP é baseado em um modelo cliente-servidor. Um ou mais servidores LDAP contém os dados criando a árvore de Diretório LDAP. Um cliente LDAP conecta-se a um servidor e faz uma requisição. O servidor responde à requisição, ou exibe um ponteiro que aponta para um local aonde o cliente pode conseguir a informação (tipicamente, outro servidor LDAP).

4.2 NIS

Um dos principais objetivos em uma rede local é fornecer para os usuários um ambiente que torne a rede transparente. Um importante passo é manter dados importantes, como informações de todas as contas de usuários na rede sincronizadas em todas as máquinas, pois isto permite ao usuário mover-se de uma máquina para outra sem o inconveniente de ter que se lembrar de diferentes senhas, ou copiar dados de uma máquina para outra.

Esse é o principal objetivo do serviço NIS (*Network Information Service* ou Serviço de Informação de Rede) [How98]. A informação administrativa que é armazenada no servidor não precisa ser duplicada, e assim é possível medir a consistência dos dados, aumentar a flexibilidade para os usuários e também tornar a vida do administrador do sistema muito mais fácil através da manutenção de uma única cópia da informação requerida [Con].

Sendo este o seu principal propósito, o servidor deve conter uma quantidade considerável de informações disponíveis na rede local, sendo que estas poderão ser:

- Nomes de acesso e senhas;
- Diretórios de usuários (ou informações do arquivo `/etc/passwd`);
- Informações de um grupo (arquivo `/etc/group`).

Se, por exemplo, uma entrada contendo uma senha é gravada em uma base de dados NIS, o usuário será capaz de acessar qualquer máquina da rede que contenha os programas-cliente do NIS que estão sendo executados, como se fosse sua própria máquina.

NIS é baseado em RPC (*Remote Procedure Call* ou Chamadas de Procedimento Remotas), e é composto basicamente do servidor, que armazena as informações, do cliente, que acessa o servidor, e de várias ferramentas administrativas [Con].

4.3 Radius/PortSlave

Para garantir um sistema ainda mais seguro existem opções de hardware e protocolos que podem garantir a autenticação, autorização e configuração das informações entre um servidor de acesso, que deseja autenticar as conexões e requisições, e um servidor de autenticação.

Radius (*Remote Authentication Dial In User Service* ou Serviço de Autenticação Remota de Usuários Dial In) é um protocolo de autenticação de usuários que permite uma maior segurança aos acessos remotos ao seu sistema [RW97]. Quando um usuário tenta acessar o sistema, um servidor de acesso faz uma requisição ao servidor de autenticação para que este valide a tentativa de acesso, retornando o resultado ao servidor de acesso. Isso permite a centralização do processo de autenticação, já que é possível ter diversos servidores de acesso usando um único servidor de autenticação central.

Portslave é um software que emula o Livingstone Portmaster II. Ele é necessário para a utilização do servidor de acesso [Con]. Na verdade, o Portslave atua como um cliente Radius, sendo utilizado para criar conexões *dial-up* aos servidores de acesso. Ele utiliza o protocolo RADIUS para autenticar um servidor remoto que contenha as informações sobre a conta do usuário.

Portanto, o Radius simplesmente permite que os acessos sejam autenticados de um servidor central (ou servidor Radius), sem a necessidade de manter as informações sobre as contas dos usuários em várias máquinas. O Portslave pode "escutar a linha"e agir como um cliente Radius, tanto para autenticação como para outros serviços como `telnet` e *shell* seguro (SSH).

4.4 Comentários Finais

O maior problema no uso de LDAP é que este é um sistema de autenticação muito complexo e lento, já o problema do NIS é a falta de segurança no trafego dos dados transmitidos entre cliente e servidor.

Capítulo 5

Proposta de Trabalho

5.1 Solução Proposta

Diante do visto até agora, decidui-se implementar um sistema remoto de autenticação e alteração de senhas de usuários. A idéia é criar um cliente PAM, que será o responsável pela autenticação e mudança de senhas dos usuários, e que o arquivo de senhas dos usuários fique no servidor de senhas, e não mais nas máquinas locais.

Desta forma, quando um usuário requisitar o *login* em uma máquina (estação), este estará enviando uma requisição ao servidor de senhas, pedindo ao servidor de senhas para autenticá-lo. Se o servidor de senhas reconhecer o usuário, mediante a validação de uma senha, este servidor devolve a estação de trabalho uma mensagem, de forma a liberar o acesso à estação a este usuário.

Para tanto, a senha digitada pelo usuário deve ser criptografada, antes de ser enviada pela rede, afim de garantir uma maior segurança do sistema, já que esta senha pode ser interceptada por alguém não autorizado. Chegando no servidor de senhas, a senha será descriptografada então o servidor de senhas tenta validar o usuário utilizando-se módulos PAM.

Além de autenticar um usuário, deve-se também permitir que este usuário mude sua senha, quando for de seu interesse. A vantagem de se ter um servidor de senhas, é permitir o usuário poder-se conectar em outra estação, imediatamente após ter mudado sua senha, e a sua nova senha já ser válida. Já que os arquivos de senhas não estão mais nas máquinas locais, e sim no servidor. Isto elimina o indesejável fato de um usuário ter mais de uma senha válida por um período de tempo (tempo de atualização dos arquivos de senhas, entre as estações).

É garantida permissão de cópia, distribuição e/ou modificação deste sistema de autenticação apresentado por esta monografia, sob os termos da Licença GPL(GNU *General Public License*) [Fou01].

5.2 Estratégia de Implementação

Para implementar este sistema de autenticação de usuários, utilizou-se *sockets* para realizar a comunicação entre as estações e o servidor de senhas, PAM para validar e mudar as senhas dos usuários e um algoritmo de criptografia para codificar a senha, para que esta senha possa ser enviada pela rede com mais segurança. A Figura 5.1 ilustra o esquema de comunicação entre estações e o servidor de senhas.

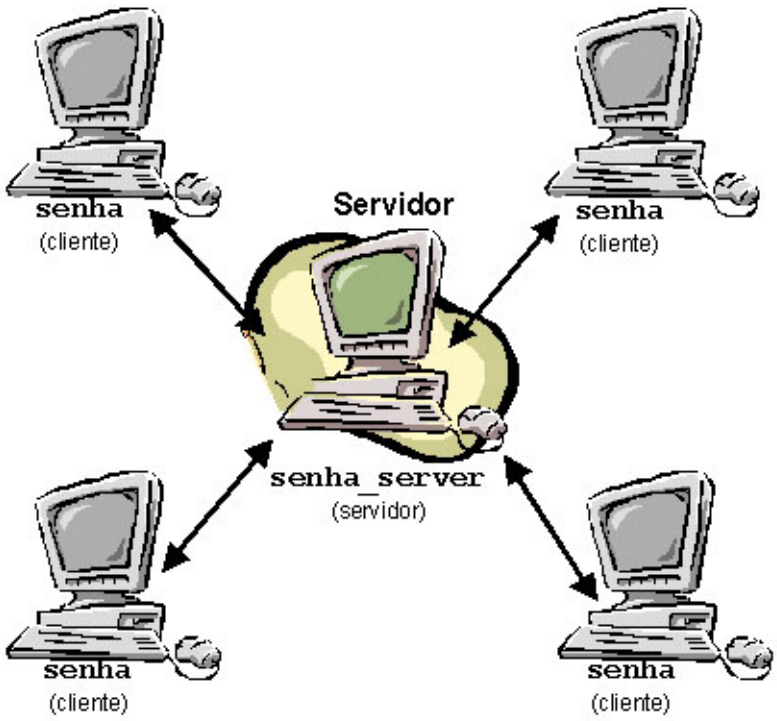

Figura 5.1: Esquema de Comunicação Estações/Servidor

Podemos verificar nesta Figura 5.1, a presença do aplicativo `senha` nas esta-

ções, as estações são os clientes na comunicação e o aplicativo `senha_server`, no servidor de senhas, o servidor de senhas é o servidor na comunicação. As setas representam a comunicação entre estes clientes e o servidor. Na seção 5.3 fala-se melhor sobre cada o aplicativo `senha` e `senha_server`.

5.2.1 Características da Implementação

Para implementação do cliente e servidor utilizou-se SSL (*Secure Sockets Layer*) para estabelecer uma comunicação segura entre os clientes (estações) e o servidor de senhas. Já que o protocolo SSL implenta algoritmos criptográficos simétricos e assimétricos e garante um comunicação segura entre cliente e servidor.

Para trabalhar com SSL você deve incluir a seguinte biblioteca em seu código fonte:

```
#include <openssl/ssl.h>
#include <openssl/err.h>
```

Ao compilar seu código você deve ligar (*link*) as bibliotecas, então passe como parâmetro para o compilador o seguinte:

```
-lssl -lcrypto
```

Você também deve gerar um certificado, uma chave pública e uma chave privada, estes devem estar sem senha, para isto você deve abrir um terminal e digitar os comandos seguintes, e responder às perguntas que serão feitas para a geração das chaves.

```
[user>] openssl req -new -text -out cert.req
[user>] openssl rsa -in privkey.pem -out server.key
[user>] openssl req -x509 -in cert.req -text -key
server.key -out server.crt
```

Para trabalhar com os Linux-PAM você precisa incluir a seguinte bliblioteca no código fonte:

```
#include <ecurity/pam_appl.h>
```

E você deve passar como parâmetro para o compilador o seguinte:

```
-ldl -lpam
```

Além destas bibliotecas você precisa de inserir outras bibliotecas para comunicação em redes que são as seguintes:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
```

A interface com o usuário é bastante simples, e não utiliza-se de recursos gráficos. Foi utilizado os recuros oferecidos pela biblioteca `ncurses.h` para programação em liguagens C e/ou C++ [Sch97] [MJ97]. Esta biblioteca é uma biblioteca para se trabalhar com janelas em modo texto, semelhante a `conio.h` utilizada em programação para DOS. Utilizou-se desta biblioteca para contrução da aplicação `senha`. Para isso você deve incluir as seguinte biblioteca em seu código fonte:

```
#include <ncurses.h>
```

Você deve "lincar"esta biblioteca ao compilar seu código fonte para isso passe para o compilador o seguinte parâmetro:

```
-lncurses
```

Maiores informações sobre a biblioteca `ncurses` pode ser encontrada em [SM99b] ou [Eri].

Tudo isto já está devidamente inserido em um arquivo *Makefile*, então para compilar os códigos fonte deste trabalho você presisa apenas digitar em um terminal o comando: `make all`. Você deve estar no diretório dos arquivos fonte.

5.3 Instalação e Configuração

Para que o sistema de autenticação funcione, você primeiramente precisa ser o administrador da rede ou ter privilégios de administrador. Após ter-se conectado como `root` no servidor de senhas você deve seguir os seguintes passos:

- Copie o arquivo de configuração `senhapam`, para o diretório `/etc/pam.d`.

Este arquivo (`senhapam`) contém as especificações necessárias para a aplicação `senha_server` carregue os devidos módulos PAM para autenticar ou mudar a senha de um usuário, a Tabela 5.1 ilustra o conteúdo deste arquivo de configuração.

```
auth      required  /lib/security/pam_stack.so  service=system-auth
account   required  /lib/security/pam_stack.so  service=system-auth
password  required  /lib/security/pam_stack.so  service=system-auth
```

Tabela 5.1: Arquivo de Configuração `senhapam`

- Copie o aplicativo `senha_server` para o diretório `/bin` no servidor de senhas. Este aplicativo é responsável por autenticar e/ou mudar a senha de um usuário (utilizando de módulos PAM) e por estabelecer uma conexão com as estações (via SSL). Este aplicativo fica a espera de uma requisição vinda de um estação. Por tanto este fica em um *loop* infinito, a espera de um contato proveniente de uma estação. Esta aplicação ficará "rodando"em *background*, para tanto ele deve ser executado com o seguinte comando (em um terminal):

  ```
  [root #]  senha_server &
  ```

  o & comercial ao final do comando é necessário para que você possa fechar seu terminal (*prompt*) ou utilizá-lo para outras coisas sem terminar a execução do aplicativo `senha`. Agora o servidor de senhas já está pronto para receber pedidos de autenticação ou mudança de senhas vindos das estações.

  Este aplicativo segue as configurações do arquivo `senhapam` que está no diretório `/etc/pam.d` para poder autenticar ou mudara senha de um usuário.

- Altere o arquivo `ssl_client.c`, você deve inserir o número do IP do servidor de senhas de sua rede, apresenta-se logo em seguida a parte do código fonte, onde deve ser mudado o valor do IP.

  ```
  // ARQUIVO CLIENT.CPP, AUTERE O VALOR DA VARIÁVEL host PARA O
  // NÚMERO DO IP DE SEU SERVIDOR DE SENHAS.

  int main( int argc, char* argv[])
  {
  ```

```
  //Você deve inserir aqui o número IP do servidor de senhas
  // número IP do servidor a ser conectado
string host = "www.xxx.yyy.zzz";
```

Substitua o valor do número IP presente na variável `host` pelo número IP do seu servidor de senhas (na ilustração acima o número IP está sendo representado pelo número $www.xxx.yyy.zzz$), para que o aplicativo `senha` saiba com quem ele deve se comunicar. Após ter mudado o valor do IP, para o IP de seu servidor de senhas compile o arquivo `ssl_client.c`. Para isso abra um terminal (*prompt*) entre no diretório onde esta o arquivo `ssl_client.c` e digite o seguinte comando no terminal:

```
[root #] make all
```

Disponibilize o aplicativo `senha` nas estações. O aplicativo `senha` é o responsável pela interface com o usuário, é ele que solicita ao usuário seu *username* e *senha*, e estabelece a comunicação com o servidor de senhas, levando o pedido do usuário, seja ele de autenticação ou de mudança de senha. Disponibilize o aplicativo `senha` no diretório `/bin`.

Apresentá-se um exemplo da interface do aplicativo `senha`, como ilustra a Figura 5.2, de um usuário, cujo *username* é `paulo`, requisitando um pedido de alteração de senha.

Para alterar sua senha este usuário digita seu *username* e logo em seguida o sistema pede sua senha, logo após é requisitado que seja digitada a nova senha e por fim pede para o usuário digitar novamente sua nova senha, afim de confirmar se estas foram digitadas corretamente pelo usuário.

Caso o usuário tenha digitado a nova senha diferente uma da outra, é retornado um erro, e este usuário deve reiniciar o processo. Caso a nova senha tenha sido digitada corretamente esta será criptografada, juntamente da senha atual, e então enviada pela rede até o servidor de senhas. Chegando no servidor de senhas, ele irá verificar se o usuário é um usuário legitimo do sistema, para então realizar a alteração requisitada.

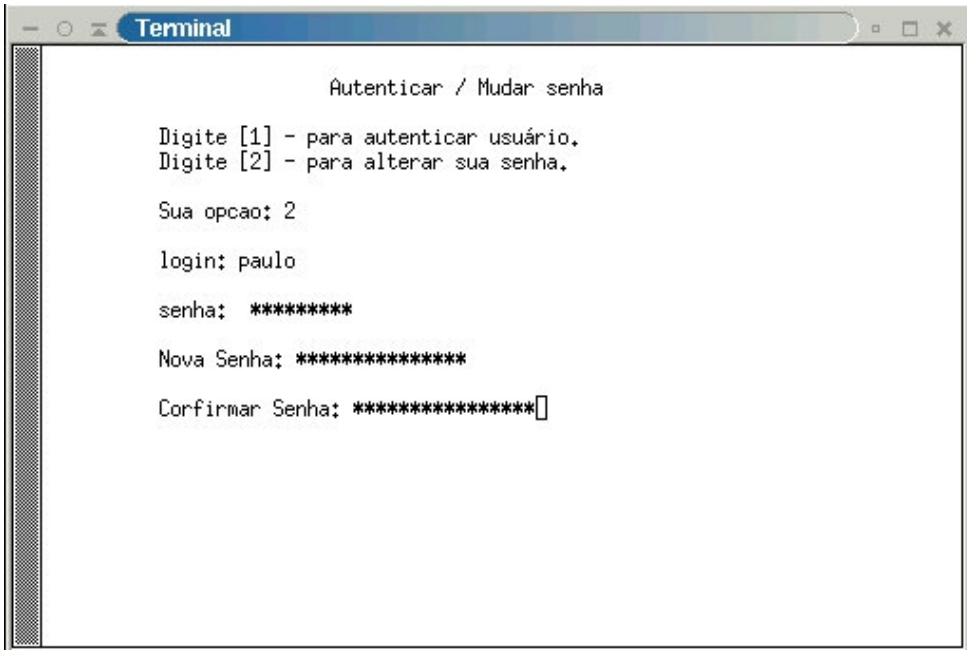

Figura 5.2: Usuário Requisitando Mudança de Senha

Capítulo 6

Considerações Finais

Este trabalho estudou técnicas de comunicação em redes *sokets*, SSL, criptografia e autenticação via módulo PAM. Apartir dos estudos realizados pôde-se desenvolver um sistema de autenticação criptografada e distribuída em estações Linux de usuários de uma rede interna.

O estudo mostrou que o sistema ao mudar a senha de um usuário, esta senha é alterada no arquivo local de senhas. E apenas de tempos em tempos todos os arquivos de senhas, distruídos pelas estações, são atualizados. Isto gera um problema de duplicidade de senhas para o usuário que requisitou o serviço de mudança de senha. Ficando este usuário impedido de utilizar sua nova senha na máquina ao lado, tendo de utilizar a sua antiga senha, até que o arquivo de senhas seja atualizado. O uso deste sistema de autenticação e mudança de senhas, elimina este problema, já que o usuário muda a sua senha no servidor de senhas e não na máquina local.

Como trabalhos futuros dando seqüência a este trabalho, sugere-se a implentação de um módulo PAM para realizar a autenticação remota de usuários, afim de substituir o módulo PAM `pam_unix.so`.

Após o termino deste módulo, deve-se focar as atenções em substituir o sistema de autenticação do NIS, já que o módulo construído trabalharia com comunicação segura.

Referências Bibliográficas

[Col00] Coutinho. Severino Collier. *Números Inteiros e Criptografia RSA*. IMPA, 2000.

[Con] Conectiva S.A. *Guia do Servidor Conectiva Linux*.

[Eri] Eric S. Raymond and Zeyd M. Ben-Halim. *Writing Programs with NCURSES*.

[Fer95] Weber. Raul Fernando. Criptografia contemporânea. *http://www.módulo.com.br*, 1995.

[Fou01] Free Software Foundation. Gnu general public license, 2001. http://www.gnu.org/copyleft/gpl.html.

[How98] L. Howard. Network information service, 1998. http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc2307.html.

[MCCI99] Steven W. Criffith Mark C. Chan and Anthony F. Iasi. *Java 1001 Dicas de Programação*, chapter Programação em Rede. Makron Books, 1999.

[MJ97] Deitel H. M. and Deitel P. J. *C++: How to Program*. Prentice Hall, 1997.

[MM00a] Scott. Mann and Ellenl. Michell. *Linux System Security the Administrato's Guide to Open Source Security Tools*, chapter Been Cracked? Just Put PAM On It! DH PTR, 2000.

[MM00b] Scott. Mann and Ellenl. Michell. *Linux System Security the Administrato's Guide to Open Source Security Tools*, chapter Cryptography. DH PTR, 2000.

[Neta] Netscape Comunications, http://wp.netscape.com/eng/ssl3/draft302.txt. *Transport Layer Security Working Group*.

[Netb] Network Associates, Inc. *An Introduction to Cryptography*, pgp*, version 6 edition.

[Oli02] Mário Luiz Rodrigues Oliveira. Uma análise da segurança e da eficiência do algorítmo de criptografia posicional. Master's thesis, Universidade Federal de Lavras, 2002.

[Ope] OpenSSL, http://www.openssl.org/. *Welcome to the OpenSSL Project*.

[Rib00] Nuno Valero Ribeiro. Computação em redes de computadores, 2000. Escola Superior de Tecnologia.

[RW97] C. Rigney and S. Willens. Remote authentication dial in user service (radius), 1997. ftp://ftp.isi.edu/in-notes/rfc2138.txt.

[SB02] Jeffrey Schiller and Steve Bellovin. Transport layer security (tls), 2002. http://www.ietf.org/html.charters/tls-charter.html.

[Sch96] Bruce Schneier. *Applied Cryptography*. John Wisley, Inc., 1996.

[Sch97] Herbert Schildt. *C Completo e Total*. Makron Books, 1997.

[SM99a] Richard Stones and Neil Mattew. *Beginning Linux Programming*, chapter Sockets. Wrox Press, 1999.

[SM99b] Richard Stones and Neil Mattew. *Beginning Linux Programming*, chapter Curses. Wrox Press, 1999.

[Sme95] Jane Smeloff. Security technology unites disparate authentication mechanisms into common infrastructure. *http://wwws.sun.com/software/solaris/pam/osf-pr.html*, 1995.

[Sun01] SunSoft. *Linux-PAM*, http://www.kernel.org/pub/linux/libs/pam/ edition, May 2001.

[Vei02] Daniel Veillard. Pam-0.64-4 rpm for i386, 2002. http://rpmfind.net/linux/RPM/redhat/updates/5.2/i386/pam-0.64-4.i386.html.

[YHK95] W. Yeong, T. Howes, and S. Kille. Lightweight directory access protocol, 1995. http://www.ietf.org/rfc/rfc1777.txt.