UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELETRÔNICA E COMPUTAÇÃO

UM MECANISMO SEGURO DE AUTENTICAÇÃO MÚTUA COM DETECÇÃO DE DESCONEXÃO E GERENCIAMENTO DINÂMICO DE REGRAS DE FIREWALL

Autor: ______________________
Bruno Astuto Arouche Nunes

Orientador: ______________________
Luis Felipe M. de Moraes, Ph.D

Examinador: ______________________
Antônio Cláudio Gómez de Sousa, M.Sc.

Examinador: ______________________
Jorge Lopes de Souza Leão, Dr.Ing.

DEL
Junho de 2004

Agradecimentos

- Aos meus pais, Miriam Cristina Astuto e Marcos Arouche Nunes, que sempre me apoiaram em tudo e me guiaram durante todas as etapas de minha vida. Seu amor, carinho e dedicação, permitiram que eu me tornasse o homem que sou hoje. Por tudo sou muito grato.
- Ao meu irmão, Rafael, simplesmente por ser o irmão que é.
- Ao meu orientador, Luis Felipe M. de Moraes, que sempre me incentivou a tentar alcançar a perfeição e por apoiar minhas idéias e permitir que elas se tornassem realidade.
- Ao professor, Marcelo Luiz Drumond Lanza, que sempre que eu precisei, esteve ao meu lado para tirar minhas dúvidas e ajudar no que eu precisasse.
- Ao grande amigo Demetrio Carrion, que junto comigo elaborou a idéia de um sistema de detecção de desligamento de estação para ser aplicado no ambiente do AirStrike. Hoje chamamos esse sistema de isAlive.
- Ao grande amigo Denilson Martins, que me ajudou quando eu não conseguia matar os *bugs* do sistema. Sua colaboração no código e sua ajuda nos testes foram de grande ajuda para a conclusão do mesmo.
- Aos funcionários e a todos os colegas do Laboratório de Redes de Alta Velocidade (RAVEL), por permitirem que as horas investidas neste laboratório fossem sempre as mais agradáveis. Agradecimento especial à Ana Elisa pela ajuda com as figuras presentes neste documento.
- Meus agradecimentos também para FAPERJ, pelo apoio ao projeto.

Resumo

Atualmente encontra-se em funcionamento no Laboratório de Redes de Alta Velocidade - RAVEL um ambiente para redes sem fio 802.11b denominado AirStrike [1], onde a segurança é o principal foco. Neste contexto, utilizou-se o sistema operacional OpenBSD e ferramentas de segurança como, redes privadas virtuais - VPNs e um sistema de controle de acesso à rede sem fio, o sistema StrikeIN. Porém, um grave problema foi identificado na implementação de ambientes de rede sem fio, que utilizam VPNs e sistemas de *firewal*. Este problema pode permitir determinados tipos de ataque.

A explicação de tal falha é simples. Uma vez que uma estação se autentica no ponto de acesso (AP), através do sistema StrikeIN, as regras do firewall no AP, são editadas, permitindo que esta estação possa usufruir as funcionalidades da rede. O problema está no fato de que, quando uma estação se desliga da rede por algum motivo, as regras do firewall continuam inalteradas, ou seja, o IP da estação que se desligou continua com permissão de acesso devido às regras do firewall. Eventuais túneis VPN que estivessem abertos com a referida estação permaneceriam abertos, mesmo que não mais necessarios, consumindo recursos da máquina funcionando como AP.

O objetivo deste trabalho é, de forma segura, promover autenticação mútua entre estações sem fio e AP, verificar se uma estação se desligou de rede e alterar automática e dinamicamente as regras do firewall, proibindo o acesso de qualquer estação com endereço IP que não tenha sido autenticado. Tendo em vista o objetivo em questão, uma aplicação cliente-servidor foi desenvolvida e implementada na linguagem C/C++. Esta aplicação leva o nome de isAlive.

Palavras-Chave

Dead Peer Detection - DPD, programação para rede, multi-plataforma, "C/C++", *Active Probing*, Redes Sem Fio, Segurança, Criptografia, Controle de Acesso.

Abreviaturas e Siglas

AP - Access Point

BD - Banco de Dados

BSS - *Basic Service Set*

CRC - *Cyclic Redundancy Check*

DPDP - *Dead Peer Detection Protocol*

DSSS - *Direct Sequence Spread Spectrum*

FHSS - *Frequency Hopping Spread Spectrum*

IBSS - *Independent Basic Service Set*

ICV - *Integrity Check Value*

IPSec - *IP Secure*

LAN - *Local Area Network*

NAT - *Network Address Translation*

pf - *Packet Filter*

PRNG - *Pseudo-Random Number Generator*

SSID - *Service Set Identifier*

STA - Estação Sem Fio

VI - Vetor de Inicialização

VPN - *Virtual Private Networks*

WEP - *Wired Equivalent Privacy*

WLAN - *Wireless Local Area Network*

Lista de Figuras

2.1 Exemplo de redes sem fio adotando topologia *ad hoc*. 7
2.2 Exemplo de redes sem fio adotando modo infraestruturado, onde um AP é utilizado na comunicação. 8
2.3 O funcionamento do WEP (extraído de [2]) 9
2.4 A autenticação no protocolo WEP (extraído de [2]) 11

3.1 Arquitetura AirStrike 16
3.2 Formato do pacote IPSec (extraído de [2]) 20

5.1 Página de autenticação para acesso ao AirStrike. 30
5.2 Troca típica de mensagens do sistema isAlive. 31
5.3 Esquema do funcionamento do isAliveDaemon. 39

Sumário

Agradecimentos **ii**

Resumo **iii**

Palavras-Chave **iv**

Abreviaturas e Siglas **v**

Sumário **vi**

1 Introdução **1**

1.1 Motivação 2

1.2 O Que Foi Feito 3

1.3 Este Documento 4

2 O Padrão IEEE 802.11b **5**

2.1 Camada Física de Redes Sem Fio 802.11b 6

2.1.1 FHSS (*Frequency Hopping Spread Spectrum*) 6

2.1.2 DSSS (*Direct Sequence Spread Spectrum*) 6

2.2 Operação de Redes Sem Fio 802.11b 7

2.3 Mecanismos de Segurança em Redes 802.11 9

2.3.1 O Mecanismo WEP (*Wired Equivalent Privacy*) 9

2.3.2 *Open System Authentication* 10

2.3.3 *Pre-Shared Key Authentication* 11

2.3.4 *Closed Network Access Control* 12

2.3.5 Listas de Controle de Acesso 12

2.3.6 Gerenciamento de Chaves 12

3 Ambiente Seguro Para Redes 802.11b - AirStrike **14**

3.1 O que é o AirStrike 15

3.2 Arquitetura de segurança proposta pelo AirStrike 16

3.2.1 Regras de *Firewall* 18

3.2.2 VPN (*Virtual Private Network*) com IPSec (*IP Secure*) 18

3.2.3 StrikeIN 19

3.2.4 isAlive 21

4 Detecção de Desligamento de Estação - DPD **22**

4.1 Importância de Mecanismos de DPD 22

4.2 Desligamento de uma Estação 23

4.2.1 *Timeout* 23

4.2.2 Sondagem Ativa (*Active Probing*) 24

4.2.3 Sondagem Passiva (*Passive Probing*) 24

4.2.4 *Keepalives vs. Heartbeats* 25

4.2.4.1 *Keepalives* 25

4.2.4.2 *Heartbeats* 26

4.2.4.3 Considerações 26

5 isAlive - Proposta, Implementação e Funcionamento **28**

5.1 Sistema de Autenticação StrikeIN 28

5.2 Protocolo DPD Proposto no isAlive 29

5.3 Implementação do isAlive 32

5.3.1 Aplicação Cliente Servidor 32

5.3.1.1 Biblioteca TCPLIB 33

5.3.2 Ambiente de Desenvolvimento 33

5.3.3 Criptografia no isAlive 35

5.3.3.1 Geração de Chaves 35

5.3.4 Acesso ao Banco de Dados 36

5.4 Funcionamento do isAliveDaemon 36

5.5 Funcionamento do isAliveStation 38

6 Instalação e Configuração do isAlive 40

6.1 Instalando o Banco de Dados MySQL 40

6.1.1 Cliente MYSQL 40

6.1.2 Servidor MYSQL 41

6.2 Instalação do isAliveDaemon 42

6.3 Instalação do isAliveStation 43

7 Conclusões e Trabalhos Futuros 45

A Código Fonte da Biblioteca TCP para Programação para Rede 47

B Código Fonte do isAliveStation 75

C Código Fonte do isAliveDaemon 89

D Script para Criação do BD StrikeIN 92

Capítulo 1

Introdução

Empresas e organizações estão rapidamente implementando infraestruturas de redes sem fio baseadas no padrão IEEE 802.11 [3], que corresponde a um padrão de redes locais sem fio atuando na faixa de 2.4 Ghz. O padrão 802.11 fornece suporte limitado à confidencialidade do tráfego neste tipo de rede, através do protocolo *Wired Equivalent Privacy* (WEP), que por sua vez apresenta falhas significantes em seu projeto [4] [5]. Além disso, o comitê do IEEE responsável pelo padrão 802.11, deixou diversas das questões de segurança mais difíceis, tais como gerenciamento de chaves e um mecanismo de autenticação robusto, como problemas em aberto, ainda esperando por uma solução. Em função disso, diversas instituições que adotam soluções sem fio utilizam chaves fixas para criptografia, ou simplesmente não utilizam criptografia alguma.

A implementação de uma rede local sem fio infraestruturada, como no caso de uma rede que adota o padrão IEEE 802.11, requer a utilização de um ponto de acesso (*Access Point* - AP) que controle as comunicações na WLAN (*Wireless Local Area Network*) e que atue como *gateway* para uma rede local (*Local Area Network* - LAN) cabeada. As WLANs transmitem dados em meio não-confinado, por utilizarem interfaces sem fio (rádio frequência) e não-delimitada por um meio condutor. Desta forma, ataques contra a autenticidade, confiabilidade e disponibilidade das transações são facilitados quando não são adotadas medidas de segurança. Isso acontece pois estas redes proporcionam risco potencial além do controle de segurança física de uma instituição.

Ao longo dos últimos anos, empresas têm empregado consideráveis esforços para proteger sua infraestrutura interna de ameaças externas. Como resultado destes esforços, essas organizações têm canalizado seu tráfego externo de rede através de "portas de acesso" distintas, protegidas por *firewall*. Essa medida, no entanto, pode não fazer efeito diante de uma rede que possua uma extensão sem fio, onde esta última, pode ser utilizada como uma *back door* para a rede interna, permitindo acesso além do perímetro de segurança física da organização. Desta forma, um atacante quando bem sucedido, pode acessar servidores da rede interna, sentado em seu carro no estacionamento da empresa (*parking lot attack*). Este cenário está descrito em [6].

1.1 Motivação

Em um esforço no sentido de solucionar estas questões em aberto e ainda sem solução, o Laboratório de Redes de Alta Velocidade - RAVEL desenvolveu e implementou um ambiente seguro para redes sem fio baseadas no padrão IEEE 802.11b, denominado AirStrike [1]. Neste contexto, utilizou-se o sistema operacional OpenBSD e algumas ferramentas de segurança como redes privadas virtuais (*Virtual Private Networks* - VPNs), *firewalls* e um sistema de controle de acesso à rede sem fio, o sistema StrikeIN. Este último, também descrito em [1]. O trabalho descrito neste documento é parte integrante deste sistema e visa solucionar alguns dos problemas já mencionados e outros que serão abordados posteriormente neste texto.

O AirStrike é uma implementação segura de um AP para redes sem fio baseado no sistema operacional OpenBSD [7] em conjunto com diversas outros softwares de código aberto sobre uma plataforma i386. No entanto, um grave problema foi encontrado na implementação deste sistema, que permitia determinados tipos de ataque. Uma vez que uma estação se autentica no AP através do sistema StrikeIN, as regras do *firewall* no AP são editadas, permitindo que esta estação possa usufruir das funcionalidades da rede. O problema está no fato de que, quando uma estação se desliga da rede por algum motivo, as regras do *firewall* continuam inalteradas, ou seja, o IP da estação que se desligou continua com permissão de acesso, uma vez

que as regras de *firewall* não foram modificadas.

O objetivo deste trabalho é, de forma segura, verificar se uma estação se desligou da rede e alterar automática e dinamicamente, as regras do *firewall*, proibindo o acesso de qualquer estação com endereço IP que não tenha sido autenticado.

1.2 O Que Foi Feito

Tendo em mente o contexto e a motivação até aqui apresentados, uma aplicação cliente-servidor foi desenvolvida e implementada na linguagem C/C++. Esta aplicação leva o nome de isAlive pois verifica se uma estações sem fio (STA) está "viva"(ligada e ativa na rede) ou não (a STA não está mais conectada a rede sem fio por algum motivo). Assim, o isAlive executa determinadas funções e procedimentos para cada caso (STA ligada ou STA desligada). O cliente isAlive, denominado aqui isAliveStation, funciona como um *daemon* na STA e aguarda requisições do servidor isAlive, chamado isAliveDaemon, executado também como um processo *daemon* no AP.

Bibliotecas do OpenSSL [8] foram utilizadas na criptografia e um mecanismo de troca de chaves foi concebido de forma a garantir autenticação mútua entre as STAs e o AP. Assim, a STA se autentica no AP, que por sua vez, também deve se autenticar na STA de forma a garantir que, por exemplo, uma STA não transmita dados para um AP forjado por um atacante.

Alguns protocolos de detecção de desligamento de estação (*Dead Peer Detection Protocol* - DPDP) foram estudados [9], mas mostraram-se ineficientes para as aplicações desejadas. Um protocolo foi desenvolvido de forma a alcançar o objetivo (detectar o desligamento de estação) e garantir segurança e eficiência ao sistema, através de trocas de chaves, mecanismos de autenticação mútua e alteração dinâmica das regras do *firewall*.

Existiu a preocupação em escrever software multi-plataforma, uma vez que há interesse em dar flexibilidade aos clientes, permitindo que as STAs possuam diferentes sistemas operacionais (Windows 98/Me/2k/XP, Linux, outros sistemas derivados do Unix). Uma biblioteca multi-plataforma foi escrita em C/C++ para auxiliar na programação para rede, de forma que toda a inicialização das conexões

TCP fossem automatizadas através da simples chamada de uma função.

1.3 Este Documento

No capítulo 2, será abordado de forma simples o padrão IEEE 802.11b e alguns fundamentos do funcionamento de redes infra-estruturadas que operam segundo esse padrão. Uma visão breve dos mecanismos de segurança previstos no mesmo também será dada neste capítulo. O capítulo 3 visa apresentar a arquitetura de segurança onde o presente trabalho se destina a ser aplicado, o AirStrike. Já o capitulo 4 apresenta uma rápida introdução sobre protocolos de detecção de desligamento de estação. Em seguida, o capítulo 5 apresenta a solução aqui proposta para o problema de detecção de desligamento de estação, bem como o funcionamento do sistema proposto e sua implementação. O capítulo 6 apresenta uma descrição detalhada do processo de instalação do software desenvolvido, tanto do cliente quanto do servidor, além de todos os passos necessários para sua configuração. Por fim, funcionalidades que podem vir a ser agregadas ao sistema e algumas mudanças no funcionamento do mesmo são mencionadas no capítulo 7. Além disso, neste último capítulo, conclusões sobre este trabalho são apresentadas.

Capítulo 2

O Padrão IEEE 802.11b

Neste capítulo será abordado o padrão IEEE 802.11b e alguns fundamentos do funcionamento de redes que o utilizam. Este padrão define também alguns mecanismos de segurança que serão abordados neste texto. As falhas já identificadas nestes mecanismos também serão estudadas.

O padrão IEEE 802.11b [3] é uma das soluções mais adotadas para redes locais sem fio (WLANs). Esse padrão está cada vez mais presente nas empresas, hotéis, fábricas e lugares públicos como aeroportos, universidades, hospitais e centros comerciais, oferecendo a possibilidade de acesso à rede com suporte à mobilidade.

O problema desta tecnologia emergente, está na sua falta de segurança, devido à particularidades do meio físico de transmissão. Como os dados são transmitidos pelo ar, não existem limites definidos como no caso das redes cabeadas. Dessa forma, é possível interceptar informações mesmo que a longas distâncias, sem necessariamente estar no mesmo ambiente ou prédio da WLAN.

As redes sem fio, geralmente, estão conectadas à infra-estrutura da rede cabeada, tornando-se assim, mais fácil para o invasor ganhar acesso a toda base de dados da empresa. Por isso, é extremamente importante a implementação de mecanismos e sistemas de segurança às WLANs.

2.1 Camada Física de Redes Sem Fio 802.11b

O padrão 802.11b lançado em 1997 pelo IEEE representa um conjunto de especificações para implementação de redes locais sem fio que prevê três técnicas de transmissão:

1. Infra-vermelho.
2. DSSS (*Direct Sequence Spread Spectrum*)
3. FHSS (*Frequency Hopping Spread Spectrum*)

Os dois últimos métodos utilizam transmissão de ondas curtas de rádio compreendidas pela banda ISM de 2.4GHz. Dentre os métodos mencionados, o DSSS é o mais utilizado na implantação de 802.11b.

2.1.1 FHSS (*Frequency Hopping Spread Spectrum*)

O FHSS utiliza 79 canais com largura de 1 MHz cada. Um gerador de números pseudo-aleatórios é utilizado para gerar a seqüência de saltos nos 79 canais. Desta forma, todas as estações que tenham utilizado a mesma semente em seu gerador e que se mantenham sincronizadas, saltarão para os mesmos canais simultaneamente. Cada estação de uma mesma rede, que utiliza a mesma seqüência de saltos, ficará em cada canal por um período denominado *dwell time*, que é ajustável. Com o FHSS tem-se um sistema robusto contra ruído de banda estreita que provê um nível de segurança já na camada física, pois somente as estações que conhecem a seqüência de saltos e o *dwell time* poderão "escutar"o meio de maneira adequada e organizada. No entanto o FHSS possui a desvantagem de oferecer uma baixa largura de banda.

2.1.2 DSSS (*Direct Sequence Spread Spectrum*)

Já o DSSS "espalha"o espectro de freqüência de um sinal de banda estreita através de sua modulação com uma seqüência de bits denominada de *chip sequence*. Obtem-se, desta forma, um sistema robusto contra ruído de banda estreita ao preço de se necessitar de um controle de potência para transmissão. Com o DSSS foi possível estender a taxa de transmissão do 802.11b de 1 Mb/s para 11 Mb/s.

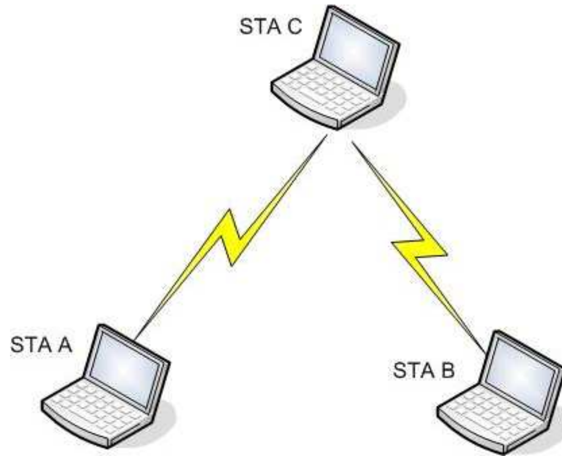

Figura 2.1: Exemplo de redes sem fio adotando topologia *ad hoc*.

2.2 Operação de Redes Sem Fio 802.11b

Redes deste tipo operam em um dos dois modos disponíveis - *ad hoc* (sem infraestrutura) ou infraestruturado. O padrão IEEE define o modo *ad hoc* como *Independent Basic Service Set* (IBSS), e o modo infraestruturado como *Basic Service Set* (BSS).

No modo *ad hoc*, cada estação pode se comunicar diretamente com outras estações na rede, como exemplificado na figura 2.1. O modo *ad hoc* foi projetado de forma que apenas as estações que se encontrem dentro do alcance de transmissão (mesma célula) umas das outras podem se comunicar. Se uma das estações (STA A) quiser se comunicar com outra fora de seu alcance (STA B), uma terceira estação (STA C) dentro do alcance de STA A e STA B, deve ser utilizada como *gateway* e e fazer o roteamento.

Já no modo infraestruturado, cada cliente envia todas as suas mensagens para uma estação central, o AP. Esta estação central fonciona como uma *ethernet bridge* e repassa as mensagens para a rede apropriada, tanto para uma rede cabeada, quanto para a própria rede sem fio, como pode ser visto na figura 2.2.

Antes de trocarem dados, STAs e APs devem estabelecer um relacionamento, ou uma associação. Apenas depois que a associação for estabelecida, as duas estações podem trocar dados. No caso de uma rede com infraestrutura, as STA se associam com o AP. O processo de associação possui dois passos, envolvendo três dos seguintes

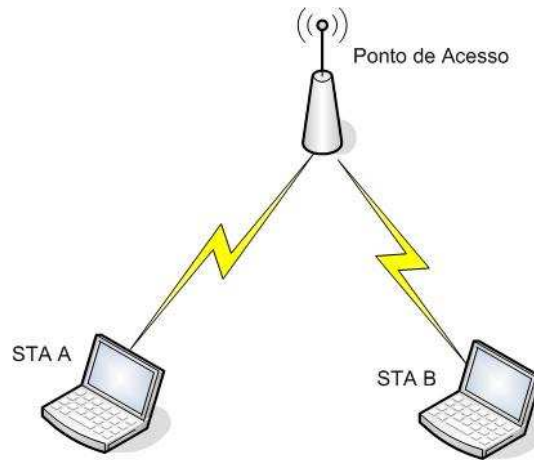

Figura 2.2: Exemplo de redes sem fio adotando modo infraestruturado, onde um AP é utilizado na comunicação.

estados:

1. Não autenticado e não associado,
2. Autenticado e não associado, e
3. Autenticado e associado.

Para mudar de estados, as partes interessadas na comunicação trocam mensagens denominadas quadros de gerenciamento ou *management frames*.

No entanto, uma STA precisa saber se existe algum AP dentro do alcance de seu rádio e a qual AP ela deve se associar. Assim, todos os APs transmitem um quadro de gerenciamento chamado *beacon* em intervalos de tempo fixos. Para se associar a um AP e se unir a uma BSS, uma estação procura escutar *beacons* para identificar APs dentro do alcance de seu rádio. Então, a STA seleciona a BSS à qual ela deseja se unir. Em seguida, AP e STA trocam diversas mensagens de gerenciamento com o objetivo de realizar autenticação mútua. Os dois mecanismos de autenticação previstos no padrão serão comentados nas seções 2.3.2 e 2.3.3.

Após uma autenticação bem sucedida, a STA passa para o segundo estado, "autenticado e não associado". Passar do segundo estado para o terceiro, "autenticado e associado", envolve a STA enviando um pedido de associação e o AP, em seguida, um quadro de resposta a esse pedido. Uma vez que esse processo ocorre, a

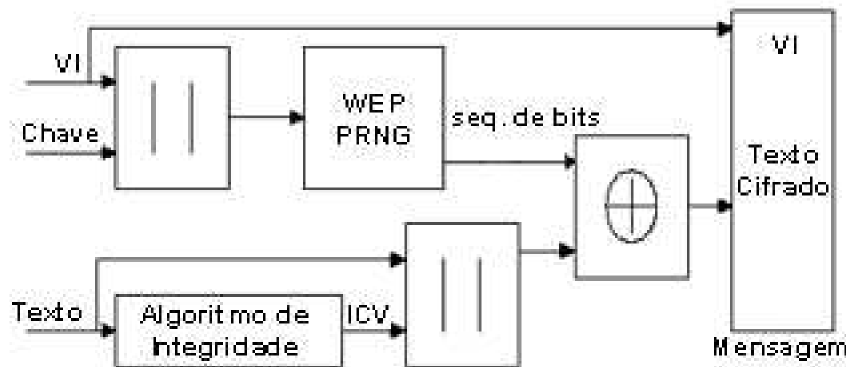

Figura 2.3: O funcionamento do WEP (extraído de [2])

STA passa a ser um ponto (*peer*) da rede sem fio e pode, então, transmitir quadros através da mesma.

2.3 Mecanismos de Segurança em Redes 802.11

O padrão 802.11 define diversos mecanismos propostos para tornar seguro um ambiente de comunicação sem fio. Esta seção apresenta a descrição de cada mecanismo.

2.3.1 O Mecanismo WEP (*Wired Equivalent Privacy*)

O protocolo WEP está definido no padrão 802.11b [3] como protocolo de segurança para as transações eletrônicas no ambiente de redes sem fio e foi projetado para garantir confidencialidade do tráfego nestas redes. Diversas falhas de segurança foram encontradas no protocolo WEP. No entanto, trabalhos como [4] e mais recentemente [5] e [10], demosntram que o WEP provê, na verdade, confidencialidade limitada.

O padrão IEEE 802.11 utiliza o protocolo WEP na camada de enlace para autenticar e criptografar os dados que serão transmitidos na rede sem fio. A figura 2.3 mostra como o WEP funciona:

O Vetor de Inicialização (VI) de 24 bits é concatenado com a chave secreta,

que pode ser de 40 ou 104 bits, resultando em uma chave composta de 64 ou 128 bits. Esta, por sua vez, serve de entrada para um gerador de números pseudo-aleatórios (PRNG) que é baseado no algoritmo RC4 [11].

A saída do PRNG é uma seqüência pseudo-aleatória de bits baseada na chave composta e com o mesmo tamanho do texto a ser criptografado. Esse texto é obtido através da concatenação do texto puro com o resultado do processo para checagem de integridade (*Integrity Check Value* - ICV) que utiliza o algoritmo de checagem de redundância CRC-32 (*Cyclic Redundancy Check*). A seqüência de bits pseudo-aleatória é utilizada para criptografar o texto através de uma operação binária XOR. O resultado do XOR é concatenado com o VI e enviado pelo emissor através do meio de transmissão.

O receptor usa o VI que vem no início do pacote e a chave secreta compartilhada para gerar a mesma seqüência criada pelo PRNG e decriptografar o texto cifrado. Então, aplica-se o CRC-32 e compara-o com o ICV concatenado ao texto puro para checar a integridade da mensagem recebida.

2.3.2 *Open System Authentication*

No mecanismo de autenticação com sistema aberto (*Open System Authentication*), a estação pode associar-se com qualquer ponto de acesso e escutar todos os dados que são transmitidos sem criptografia. Este método basea-se na transmissão da identidade da estação que quer ser autenticada para a estação que realizará a autenticação;

É o protocolo de autenticação padrão para 802.11. Como o próprio nome sugere, este sistema autentica qualquer estação que a esteja requisitando. Ou seja, não é um processo muito eficiente, do ponto de vista de quem se preocupa com segurança. Experimentos mostram [6] que estas requisições (os quadros de gerenciamento) são enviadas abertamente (sem criptografia), mesmo quando o WEP está habilitado.

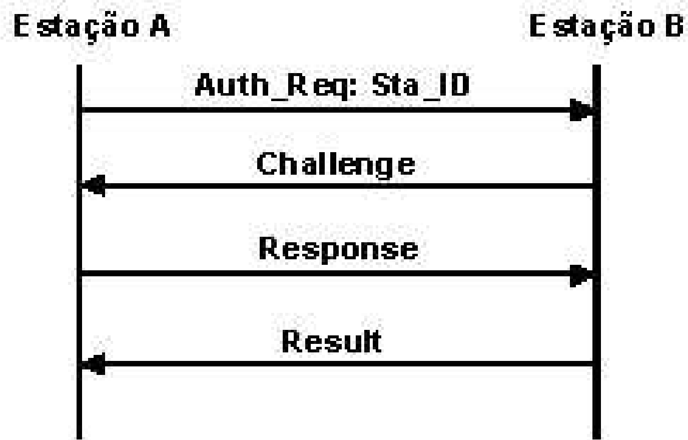

Figura 2.4: A autenticação no protocolo WEP (extraído de [2])

2.3.3 *Pre-Shared Key Authentication*

Autenticação com chave pré-compartilhada (*Pre-Shared Key Authentication*) utiliza um processo desafio-resposta em conjunto com uma chave secreta compartilhada, de forma a realizar o processo de autenticação. A STA que deseja se autenticar, quem inicia o processo, envia um quadro de gerenciamento contendo o pedido de autenticação, indicando que se deseja usar *shared key authentication*. O AP, quem responde, envia um quadro de gerenciamento contendo a resposta ao pedido de autenticação, formada por 128 octetos de texto, como desafio para a STA. O texto do desafio é criado pelo gerador de números pesudo-aleatórios (*pseudo-random number generator*- PRNG) do WEP, junto com o *shared secret* e o vetor de inicialização (VI). Uma vez que a STA recebe o quadro de gerenciamento contendo o desafio, ela copia o conteúdo do texto deste desafio para o novo quadro de gerenciamento. Este novo quadro é encriptado com WEP utitlizando a chave compartilhada, juntamente com um novo VI selecionado pela STA. O quadro de gerenciamento encriptado é então, enviado para o AP, que decripta este quadro recebido e o submete ao ICV usando 32-bit CRC. Se o ICV for válido e o texto do desafio for o mesmo enviado na primeira mensegem, a autenticação será bem sucedida. Caso a autenticação seja bem sucedida, AP e STA trocam de papeis e repetem todo o processo para garantir autenticação mútua.

A figura 2.4 ilustra este processo de autenticação. A estação A envia uma

requisição e a sua identificação (Auth_Req; Sta_ID) para a estação B. B responde com uma mensagem contendo um desafio de 128 bits (*Challenge*). A estação A copia o desafio em uma nova mensagem, criptografa com a chave WEP pré-compartilhada e reenvia para B (*Response*). A estação B checa a resposta de A e responde com o resultado do procedimento de autenticação (*Result*).

2.3.4 *Closed Network Access Control*

A Lucent definiu um mecanismo de controle de acesso proprietário chamado *Closed Network* [12]. Com esse mecanismo, é possível usar tanto redes abertas quanto fechadas. Em uma rede aberta, qualquer um tem permissão de se associar à rede. Em uma rede fechada, apenas os usuários cientes do nome e SSID da rede podem se associar à mesma. Na verdade, pode se dizer que o nome da rede funciona como uma chave pré-compartilhada.

2.3.5 Listas de Controle de Acesso

Outro mecanismo utilizado por vendedores (mas não definido no padrão) para promover segurança no acesso a rede é o uso de listas de acesso baseadas nos endereços MAC dos clientes (*Access Control Lists*). Cada AP pode limitar o acesso de clientes sem fio à rede, aos que possuirem seus endereços MAC na lista. Caso o endereço MAC de um cliente não esteja na lista, seu acesso é negado.

Este tipo de restrição não garante que apenas máquinas autorizadas (cujos endereços MAC das interfaces estejam cadastrados na lista) possam acessar a rede, uma vez que técnicas de MAC *spoofing* são amplamente conhecidas.

2.3.6 Gerenciamento de Chaves

Gerenciamento de chaves no padrão 802.11 é um outro aspecto deixado a cargo de desenvolvedores e fabricantes. Por esse motivo, apenas alguns dos grandes fabricantes implementaram, em seus produtos, alguma forma de gerenciamento de chave. No entanto, em alguns casos, os detalhes disponíveis no equipamento indicam um problema ainda maior pois eles revelam a utilização de protocolos que possuem

vulnerabilidades amplamente conhecidas.

O 802.11, no entanto, fornece dois métodos para utilização de chaves WEP. O primeiro oferece uma janela de quatro chaves registradas de forma manual. A estação ou AP pode decriptar pacotes emcriptados com qualquer uma das quatro chaves. A transmissão, no entanto, está limitada à chave padrão, que nada mais é do que uma das quatro chaves já mencionadas. O segundo método é chamado de tabela de mapeamento de chaves. Neste método, cada endereço MAC pode possuir sua própria chave separada. O tamanho desta tabela deve ser de, no mínimo, dez (10) entradas de endereços de acordo com as especificações do 802.11. O tamanho máximo, por sua vez, depende do *chip-set* utilizado pelo fabricante. A utilização de chaves separadas para cada usuário, ameniza problemas relacionados a ataques criptograficos, facilitados por outros métodos. O problema deste tipo de mecanismo reside em encontrar um período de chave razoável, uma vez que as chaves são trocadas manualmente.

Capítulo 3

Ambiente Seguro Para Redes 802.11b - AirStrike

A implementação de uma WLAN infraestruturada requer a utilização de um ponto de acesso (AP) que controle as comunicações na WLAN e que atue como *gateway* para uma LAN cabeada ou à Internet. Juntamente com a disseminação e utilização deste padrão surgiram diversas questões relacionadas à segurança das transações eletrônicas que notoriamente encontram novos desafios em uma interface aérea e não-delimitada por um meio condutor.

A arquitetura da WLAN é outro aspecto fundamental para a garantia de segurança dos usuários móveis, do AP e da própria infraestrutura da rede local cabeada de uma entidade, que tem na WLAN uma extensão de sua abrangência. A utilização de *firewalls*, endereços IPs não roteáveis, por exemplo, permitem que o tráfego da rede sem fio seja contido e controlado. A entidade central na implementação de uma WLAN é o AP, portanto oferecer meios que permitam implementar um AP com baixo custo e com total controle sobre as características de segurança dos serviços e softwares configurados no mesmo é um dos objetivos do AirStrike.

Desta forma, tanto na instalação do AP quanto na implementação do isAlive, optou-se pela utilização de softwares de código aberto e gratuitos, contando assim com uma ampla documentação e suporte de diversos desenvolvedores na comunidade de código aberto. Esta seção descreve a implementação de uma solução de segurança para WLAN baseada no sistema operacional OpenBSD [7]. Para tal, utilizou-se

um microcomputador como AP onde realiza-se autenticação mútua, autorizando a comunicação entre as estações da rede sem fio e o AP através do fornecimento de credenciais como login, senha e certificados digitais. Além disso, criptografando-se toda a comunicação entre as estações da WLAN e o AP de forma a garantir a confidencialidade.

3.1 O que é o AirStrike

O AirStrike é uma distribuição do sistema operacional OpenBSD que permite a configuração de um AP a partir de um microcomputador, de forma que sejam instalados por padrão um conjunto de ferramentas de segurança que garantam a autenticação dos usuários móveis da WLAN e o sigilo de todas as transações eletrônicas entre as estações da WLAN e o AP.

O OpenBSD é um sistema operacional UNIX gratuito, baseado no BSD 4.4, cujos principais esforços enfatizam a portabilidade, padronização, correção, segurança proativa e criptografia integrada.

O AirStrike estabelece também uma arquitetura de segurança para utilização de uma WLAN como extensão da abrangência de uma LAN cabeada, permitindo o controle e gerência de vários APs a partir de uma estação de gerência. Os objetivos do AirStrike são:

1. Desenvolver uma distribuição do sistema operacional OpenBSD, denominada AirStrike, que capacite um usuário a configurar de forma simples e rápida um computador como AP.

2. Gerar um conjunto de sugestões e práticas que contemple a segurança da interconexão de WLANs em ambientes com infraestrutura cabeada préexistente.

3. Implementar um sistema de autenticação de usuários baseado em certificados digitais e credenciais fornecidas por login e senha.

4. Configurar uma VPN entre as estações WLAN e o AP.

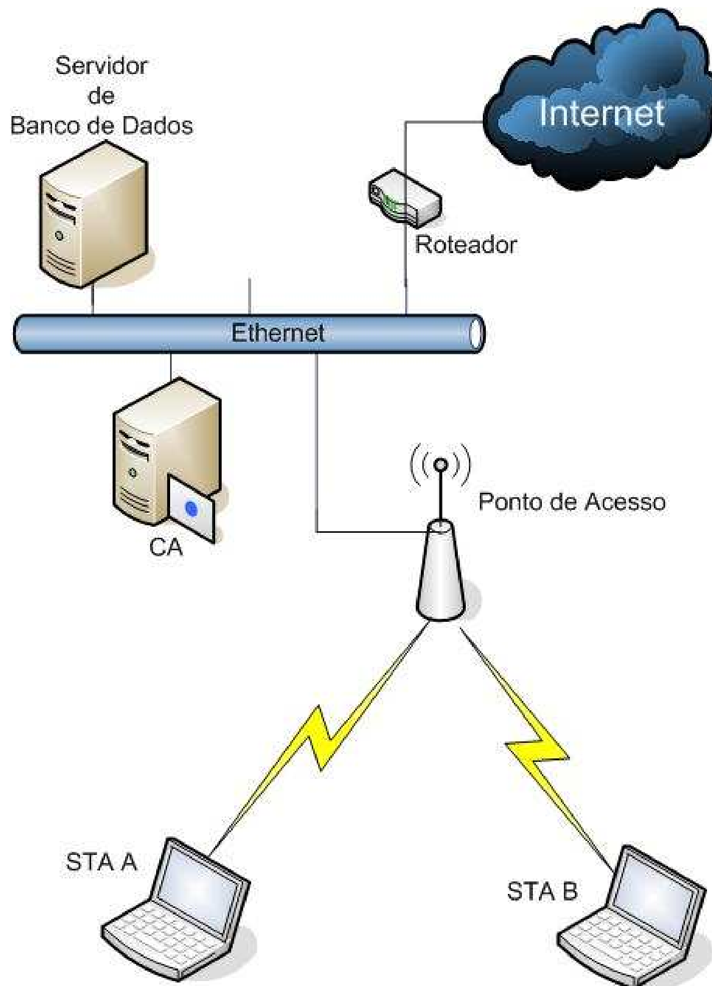

Figura 3.1: Arquitetura AirStrike

3.2 Arquitetura de segurança proposta pelo AirStrike

Diversas medidas de segurança são adotadas na configuração do AP AirStrike. Algumas considerações de segurança são relativas à própria configuração segura de um sistema operacional e outras são relativas aos mecanismos de autenticação, autorização, confidencialidade e integridade do fluxo de informações presentes na WLAN gerenciada pelo AP. Uma arquitetura proposta para a integração de uma WLAN a uma LAN cabeada está representado na figura 3.1.

Está descrito abaixo o processo pelo qual uma STA passa desde o momento em que deseja se associar ao AP até o fechamento da conexão. É importante entender tal procedimento para que fique claro o papel do sistema isAlive, foco deste trabalho, onde ele atua e que problemas o mesmo se propõe a resolver.

1. STA procura uma rede no domínio definido pela SSID
2. STA e AP se sincronizam e a associação é estabelecida
3. STA requisita um IP (cliente DHCP)
4. AP fornece um IP à STA (servidor DHCP)
5. STA envia mensagens UDP para formação do túnel VPN
6. AP verifica credenciais do usuário móvel, formando o túnel VPN
7. Usuário da STA acessa página web a fim de se autenticar
8. O AP requisita o certificado digital do usuário da STA
9. O usuário da STA apresenta o seu certificado digital
10. O AP apresenta o seu certificado digital à STA
11. O AP acrescenta o IP da STA no banco de dados de IPs autorizadas
12. O AP reconfigura as regras de *firewall*
13. A STA está pronta para utilizar os recursos de rede, oferecidos através do AP
14. Durante o período de conexão, o AP verifica se a STA continua ativa, a fim de que possa controlar de forma adequada as regras do *firewall*. Esta e outras tarefas ficam a cargo dos sistema isAlive, descrito no capítulo 5

A segurança do ambiente AirStrike está dividida em quatro partes fundamentais. Cada uma destas quatro partes será descrita nas seções subsequentes. São elas:

1. Regras de *Firewall*, seção 3.2.1
2. VPN, seção 3.2.2
3. StrikeIN, seção 3.2.3
4. isAlive, seção 3.2.4

3.2.1 Regras de *Firewall*

O OpenBSD possui um filtro de pacotes denominado pf (*packet filter*) que permite a configuração de um *firewall* no AP, assim como a utilização de endereços não-roteáveis através da aplicação de NAT (*Network Address Translation*) [13]. Desta forma pode-se controlar o tráfego presente na WLAN através da construção de regras de *firewall* específicas e dificulta-se, através do uso do NAT, o descobrimento das máquinas existentes em uma WLAN para um inimigo que se encontre fora do ambiente definido pela WLAN.

O uso do pf é vital para o funcionamento e controle de todo o ambiente AirStrike pois, a mudança dinâmica destas regras permite que um usuário móvel tenha acesso seletivo aos recursos da rede na medida em que ele forneça as credenciais necessárias para passar para o próximo nível de acesso aos mesmos.

Inicialmente as regras de *firewall* só permitem que o tráfego destinado ao estabelecimento de uma VPN seja transmitido entre a STA e o AP. Após o estabelecimento da VPN permite-se que deste tráfego criptografado, pelo protocolo ESP, somente uma página web de autenticação presente no AP possa ser acessada, onde credenciais como certificados digitais (autenticação mútua) e o par login/senha (autorização) são exigidos. Após a verificação e validação do par login e senha em um banco de dados MySQL, implementado e hospedado em uma máquina específica da LAN cabeada, modificam-se mais uma vez as regras do *firewall*, permitindo que este usuário da WLAN acesse todos os recursos de rede oferecidos pelo AP através da VPN.

É fundamental verificar se a estação da WLAN continua ativa ao longo da utilização dos recursos do AP de forma a se previnir roubo de sessão e garantir a eficácia das regras do *firewall*. Isto é garantido pelo isAlive que sonda ativamente a presença de uma estação na rede, como será visto mais a frente.

3.2.2 VPN (*Virtual Private Network*) com IPSec (*IP Secure*)

O sigilo das transações eletrônicas no ambiente AirStrike é garantido pela implementação de uma VPN entre o AP e as estações WLAN, sendo assim, todo o tráfego encontra-se critptografo no meio de transmissão.

Dentre várias propriedades desejáveis em comunicação de dados segura, pode-se citar: a confidencialidade, a integridade e a autencidade, como as mais comuns. Mecanismos eficientes para garantir confidencialidade dos dados em redes sem fio são as VPNs. Aqui será abordado rapidamente o protocolo IPSec para estabelecer a VPN entre o ponto de acesso e as estações sem fio. O IPSec provê segurança através de criptografia e/ou autenticação na camada IP.

O IPSec fornece estas características na utilização de dois protocolos: Cabeçalho de Autenticação (*Authentication Header - AH*) [14] e o Encapsulamento Seguro do Campo de Dados (*Encapsulating Security Payload - ESP*) [15]. O AH e o ESP suportam dois modos de operação: transporte e túnel.

- **Modo Transporte**: O modo transporte provê proteção para os protocolos da camada superior (TCP e UDP, por exemplo). É usado, geralmente, em comunicações fim-a-fim entre estações. Neste modo o ESP criptografa e, opcionalmente, autentica o campo de dados do pacote IP. O AH autentica o campo de dados IP e campos do cabeçalho IP. A Figura 3.2(b) ilustra o ESP e o AH no modo transporte.

- **Modo Túnel**: Tipicamente utilizado para estabelecer VPNs, provê proteção ao pacote IP completo. Os campos do AH e ESP são adicionados ao pacote IP e tudo passa a ser tratado como o campo de dados do pacote IP de um novo pacote, inclusive com um novo cabeçalho IP. A Figura 3.2(c) mostra o funcinamento do IPSec em modo túnel. O ESP criptografa e, opcionalmente, autentica todo o pacote IP interno. O AH autentica o pacote IP interno e partes do cabeçalho externo.

3.2.3 StrikeIN

O processo de autenticação está intimemente ligado com o funcionamento do isAlive. Assim, o sistema StrikeIN será abordado mais adiante no texto na seção 5.1.

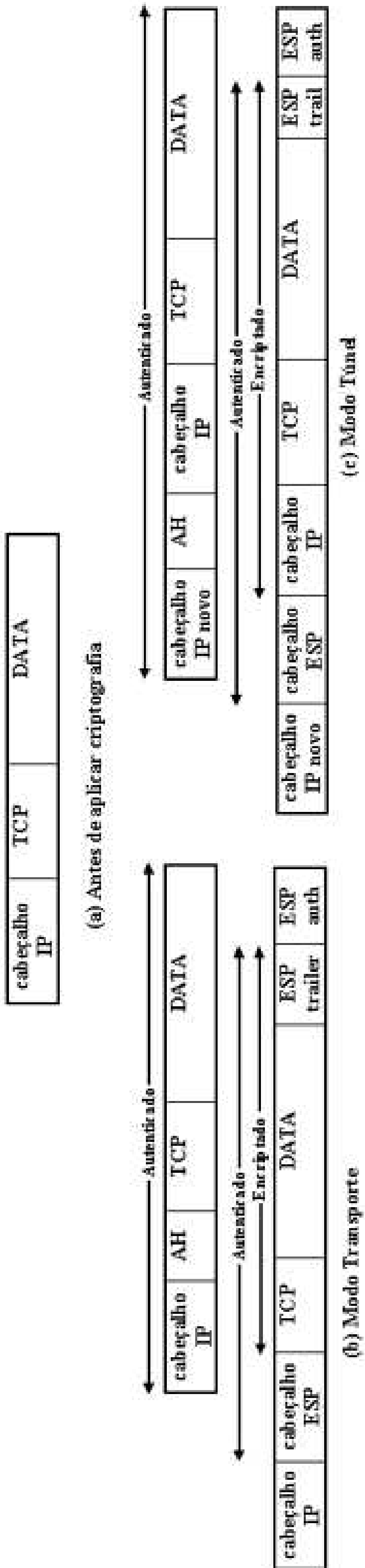

Figura 3.2: Formato do pacote IPSec (extraído de [2])

3.2.4 isAlive

A implementação e o funcionamento do isAlive e seu protocolo DPD, serão descritos em seções subsequentes no capítulo 5.

Capítulo 4

Detecção de Desligamento de Estação - DPD

Este capítulo se dedica a introdução do tema envolvendo mecanismos de *Dead Peer Detection* - DPD, sua importância e relevância neste trabalho, bem como a apresentação de alguns destes mecanismos propostos na literatura. Tais mecanismos foram estudados e analisados ao longo deste trabalho e algumas conclusões e críticas a tais mecanismos também podem ser encontradas nas seções deste capítulo.

4.1 Importância de Mecanismos de DPD

Quando existe comunicação entre dois pontos de rede (mais especificamente no caso em estudo, entre o AP e a STA), poderá existir uma situação onde a conexão entre eles será desfeita de forma inesperada. Esta situação pode ocorrer devido a problemas de roteamento, reinicialização de uma das partes comunicantes, a STA sai da área de cobertura do AP, a bateria da STA acaba, entre outros fatores. Em casos como estes dificilmente existe um meio do AP identificar esta perda de conectividade.

Em situações assim, as regras de *firewall* permaneceriam inalteradas e as permissões de acesso dadas ao IP da STA recém desconectada ainda estariam garantidas. Além disso, no caso em que se utiliza túneis IPSec entre STA e AP, o túnel destinado a STA que se desconectou continuaria ativo e consumindo recursos do AP.

Problemas de detecção de desconexão têm sido endereçados por propostas que

necessitam o envio de mensagens periódicas de HELLO/ACK como confirmação de que a estação está ainda "viva". Esses esquemas tendem a ser unidirecionais (apenas envia de mensagens de HELLO) ou bidirecionais (envio do par HELLO/ACK).

Segundo [9] o problema com este tipo de esquemas com envio periódico de mensagens está, justamente, no fato de existir a necessidade de se enviar as mesmas em intervalos regulares de tempo. Na implementação, isto se traduziria no gerenciamento de um relógio responsável por contar o tempo entre mensagens.

4.2 Desligamento de uma Estação

A autenticação e autorização de uma estação implica em mudanças nas regras do *firewall*, desta forma deve-se prover um mecanismo para mudança das regras quando a mesma estação se desliga da rede.

Não é prudente contar com uma mensagem de requerimento de dissociação da rede por parte de uma STA, pois os motivos de sua saída podem ser os mais diversos (perda de sinal, interferência, falta de energia, etc), motivos estes que a impediriam o envio de uma mensagem deste tipo.

4.2.1 *Timeout*

Uma proposta de fácil implementação seria a de se adotar um tempo de conexão limitado para as estações, requisitando uma reautenticação após este tempo ter decorrido. Isto implica em um grande inconveniente para os usuários, que deverão se reautenticar diversas vezes ao longo de uma conexão. A escolha deste intervalo de conexão estaria limitada pela falta de funcionalidade e praticidade, que um curto período de tempo traria ao usuário e a diminuição na segurança da rede para usuários que se desconectassem antes deste tempo chegar ao fim, uma vez que as regras do *firewall* estariam abertas para aquele IP.

Fica claro, portanto que propostas mais avançadas devem ser empregadas. Algumas delas estão descritas em seguida.

4.2.2 Sondagem Ativa (*Active Probing*)

O procedimento de sondagem ativa envolve algum tipo de mensagem enviada à estação sem fio, assim como nos mecanismos uni e bidirecionais mencionados anteriormente e descritos em maiores detalhes na seção 4.2.4. Um processo como o envio de um *ping* para o cliente, por exemplo, é uma forma de se verificar se uma estação cliente continua ativa na rede, porém algumas considerações devem ser feitas.

O envio de um ping não previne que um cliente válido se desligue da rede e que em um instante subseqüente uma estação, controlada por um atacante e utilizado o mesmo IP, entre na rede. Desta forma a estação inimiga roubaria a seção da estação válida, se fazendo passar pela mesma.

Uma proposta intrínseca à construção do projeto AirStrike se baseia no fato de que após o processo de autenticação, descrito ao longo da seção 5.1 ser completado, assume-se que toda a comunicação entre o AP e a estação passa por uma VPN e portanto uma estação inimiga que roubasse a seção da estação válida não teria meios de decriptar o *probe* representado pelo ping e não teria como respondê-lo de forma adequada.

Uma terceira proposta, independente da configuração de uma VPN, seria a instalação de uma espécie de *plugin* no cliente que responde a uma requisição de um *daemon* presente no AP. Este mecanismo seria semelhante a um ping, mas neste caso, informações adicionais seriam trocadas de forma a garantir a autenticidade da estação.

4.2.3 Sondagem Passiva (*Passive Probing*)

O *firewall* contido no AP possui uma tabela de estados das conexões que um cliente mantém ativas. Um mecanismo poderia ser implementado, para que fosse verificado se um determinado cliente está ou não registrado na tabela de estados do *firewall*. Caso ele não estivesse, isto serviria como indicativo de que a conexão foi desligada. Algumas observações podem ser feitas a este respeito. Primeiramente, recairia-se no caso do ping trafegando em uma rede sem criptografia, ou seja, um cliente válido que se desligasse, seguido da entrada de um inimigo, indicariam a

este mecanismo proposto que o cliente continua ativo. Esta caso caracterizaria falha de segurança, onde as regras do *firewall* não seriam capazes de impedir a ação do atacante.

O uso da sondagem ativa requer cuidados com relação a utilização da banda da rede e na definição do período entre verificações de cada cliente e para cada cliente. O uso de sondagem passiva conta somente com o processamento no AP e não consome banda, no entanto sofre da falha do roubo de uma sessão por um inimigo.

4.2.4 *Keepalives vs. Heartbeats*

Nesta seção, será feita uma análise dos mecanismos Keepalive e Heartbeat, descritos em [9]. Nesta referência tais mecanismos são descritos através de exemplos de comunicação entre dois pontos em uma rede sem fio, *peer* A e *peer* B. Neste texto, tal análise será realizada do ponto de vista da arquitetura aqui estudada, ou seja, a arquitetura de redes sem fio infraestruturada utilizada no sistema AirStrike (como visto nas figuras 2.2 e 3.1).

4.2.4.1 *Keepalives*

Considere um sistema de keepalive onde um AP requer, regularmente, indicações de que uma determinada STA está conectada à rede e vice-versa. Os dois lados da conexão (AP e STA) entrariam em um acordo sobre o intervalo entre as mensagens de keepalive, o que significa que deve existir algum tipo de negociação durante a fase inicial de comunicação. Para que a detecção de desconexão seja possível, as mensagens de keepalive devem ser enviadas freqüentemente, por exemplo, a cada intervalo de tempo T segundos. Essencialmente, a cada T segundos, um deve enviar uma mensagem de HELLO para o outro. Esta mensagem serve, para quem a envia, como uma prova de que o outro ponto de comunicação (quem a recebe) está "vivo". Em troca, o outro ponto deve admitir recebimento de cada HELLO com uma mensagem de ACK. Deve existir em cada estação comunicante (AP e STA), um contador de tempo. Se T segundos se passaram, e um dos lados (o AP por exemplo) não recebeu o HELLO, este irá enviar uma mensagem de HELLO

para a STA. A decorrente mensagem de ACK gerada pela STA serve como prova de vida da mesma para o AP. O recebimento tanto de mensagens HELLO quanto de ACKs fazem com que o contador recomece a contagem do tempo decorrido entre mensagens. Falha no recebimento de um ACK durante um determinado período de tempo, caracteriza um erro. Após um determinado número de erros considera-se que a estação não está mais conectada à rede.

Neste tipo de esquema, a parte interessada (AP) em detectar se o outro ponto de rede (STA) está conectado é quem inicia a troca de mensagens. É concebível, em tal esquema, que a STA nunca esteja interessada em saber se o AP esta vivo ou não. Nesse caso, o ônus cairia sempre sobre o AP em iniciar a comunicação.

4.2.4.2 *Heartbeats*

Nesta seção, será considerado um esquema de DPD envolvendo mensagens unidirecionais. A entidade interessada na situação de uma outra estação, dependeria do envio de mensagens periódicas por parte desta, de forma que, tais mensagens demonstrem que esta determinada estação está viva. Assim, se o AP quiser saber se uma STA está conectada, a mesma deverá enviar a cada intervalo de tempo T, uma mensagem HELLO para o AP. Caso este não receba a mensagem de HELLO esperada, proveniente da STA, será computado um erro. Mais uma vez, após um determinado número de erros, considera-se que a estação não está mais conectada à rede.

Uma das desvantagens neste tipo de esquema de DPD é confiar na STA para enviar as mensagens que demonstram uma conexão ativa. Apesar disso, se o AP quiser saber se a STA está viva, a estação deve estar ciente disso. Fica claro que, também neste cenário, um período de negociação é necessário antes do início da sondagem.

4.2.4.3 Considerações

Essencialmente *heartbeats* e *keepalives* baseiam-se em trocas de mensagens de HELLO em intervalos regulares. Em [9], os autores afirmam que, em um esquema de DPD, o estado de uma estação não pode estar dependente do estado de outra

e que a prova de vida de um ponto de rede para outro só deve ser enviada quando necessário e não determinada por intervalos de tempo. Afirma-se ainda que estas mensagens devem ser apenas trocadas quando o canal estiver ocioso, pois o próprio tráfego entre as estações serviria como prova de vida de uma para a outra.

Considere agora, novamente, um cenário onde existe um *firewal* em um AP, cujas regras necessitam ser editadas dinamicamente e com urgência (no momento em que um evento ocorre a regra do *firewall* deve ser alterada o mais rápido possível) e uma STA que conecta-se à rede através deste AP. O AP precisa receber informações periodicamente a respeito do estado desta estação. Existe um problema, no entanto, caso as mensagens de HELLO sejam enviadas apenas em momentos de silêncio (não existe tráfego proveniente de STA). Se este tráfego for forjado por um atacante, quando a STA se desconectar da rede, a STA inimiga continuará enviando pacotes para o AP em nome da STA e o AP, em momento algum irá requisitar prova de vida desta estação, uma vez que a STA (do ponto de vista do AP) ainda está enviando tráfego. As regras de firewall permaneceriam inalteradas para que o atacante pudesse usufruir das funcionalidades da rede.

Em [9], o protocolo DPD proposto executa requisições de prova de vida apenas quando não existe comunicação entre as estações e somente quando uma das estações está interessada no estado (conectado ou não) da outra. No cenário estudado neste trabalho (figura 3.1), o AP sempre estará interessado no estado da STA. Este cenário e a proposta de implementação de um protocolo DPD aplicado ao mesmo serão apresentadas nas seções posteriores.

Capítulo 5

isAlive - Proposta, Implementação e Funcionamento

Neste capítulo, será apresentada a proposta de um protocolo DPD implementado pelo isAlive e utilizado no sistema AirStrike dentro de um ambiente infraestruturado de rede sem fio, como pode ser visto na figura 3.1. Neste capítulo, detalhes da implementação serão abordados, onde todo o funcionamento do sistema isAlive será apresentado.

5.1 Sistema de Autenticação StrikeIN

Uma STA que queira fazer parte de uma rede sem fio local deve possuir um IP para poder se comunicar com os demais membros da rede. Na implementação de redes WLAN, que tem como característica um certo nível de mobilidade das estações, torna-se mais interessante prover estes IPs de forma dinâmica, utilizando-se do DHCP. Portanto, uma STA que já tenha se associado a uma BSS[1] irá requisitar um IP do AP de forma a poder enviar e receber informações através do mesmo. Este IP é fornecido à estação, mas alguns procedimentos a mais devem ser completados antes da estação poder enviar e receber informações através da rede.

Existe, no AP, um *firewall* que bloqueia ou permite acesso aos recursos da

[1] *Basic Service Set* ou Unidade Básica de Serviço se assemlha à uma célula de comunicação formada ao redor de uma Estação Base existente na telefonia celular.

rede de acordo com procedimentos de autenticação, detalhados a seguir nesta seção. Algumas ferramentas *OpenSource* foram desenvolvidas por terceiros com este objetivo, mas nenhuma delas mostrou-se adequada para este projeto. Podem ser citadas:

1. **Oasis** - Ferramenta de autenticação desenvolvida por um grupo *StockholmOpen.net*, cujo principal empecilho em sua utilização é o fato de se destinar a plataformas Linux e FreeBSD e não é suportado no OpenBSD [16].

2. **NoCAT** - Projeto desenvolvido por uma comunidade de usuários de redes 802.11. Esta ferramenta roda em plataforma Linux e só verifica a saída de um cliente da rede através de timeout [17].

3. **NetLogon** - Roda em plataforma Linux e não faz autenticação em banco de dados [18].

No StrikeIN, o usuário, após ter se associado ao AP (ver passo-a-passo na seção 3.2), deve acessar uma página de autenticação (figura 5.1) e entrar com seu login e sua senha. Uma vez autenticado, o endereço IP utilizado por aquele usuário e sua senha, são armazenados em uma tabela no banco de dados chamada *IPsValidos*. Esta tabela será utilizada pelo isAlive para a detecção segura de desligamento da STA.

É importante que antes que o usuário faça o login no sistema ele inicie o isAliveStation (ver seção 5.5).

5.2 Protocolo DPD Proposto no isAlive

Foi proposto um mecanismo de detecção de desligamento de estação baseado em sodagem ativa das mesmas. Este tipo de sondagem de estações foi descrito na seção 4.2.2. Detalhes do protocolo proposto e das trocas de mensagens entre STAs e o AP podem ser vistos na figura 5.2.

Para iniciar o processo de detecção de desligamento de estação, o AP gera uma seqüência de bytes aleatórios (PKT-KEY1) e a encripta utilizando a senha (PWD) do usuário cadastrado como chave. A seqüência encriptada é enviada à STA que a decifra utilizando a chave pré-compartilhada (a senha do usuário, PWD).

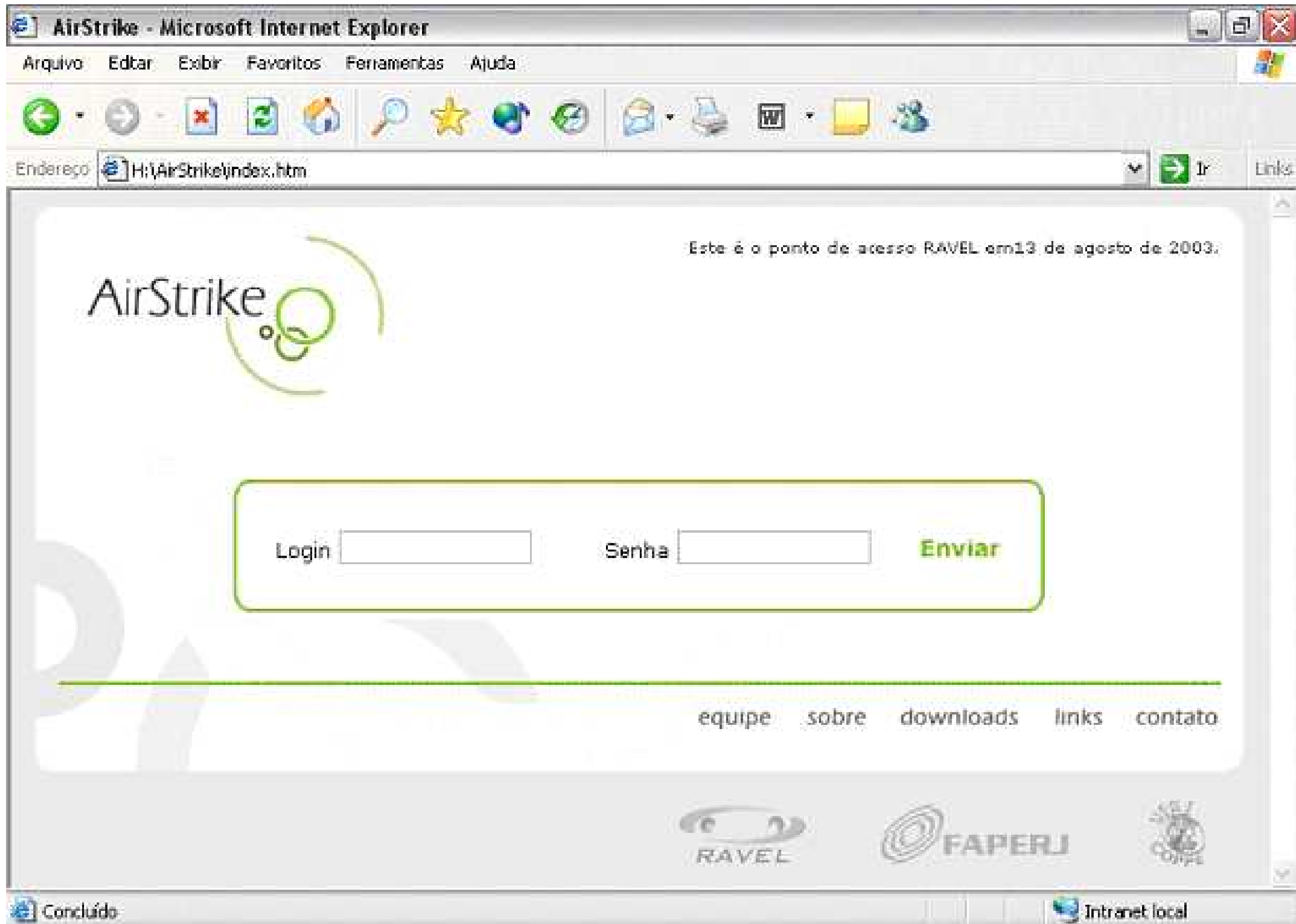

Figura 5.1: Página de autenticação para acesso ao AirStrike.

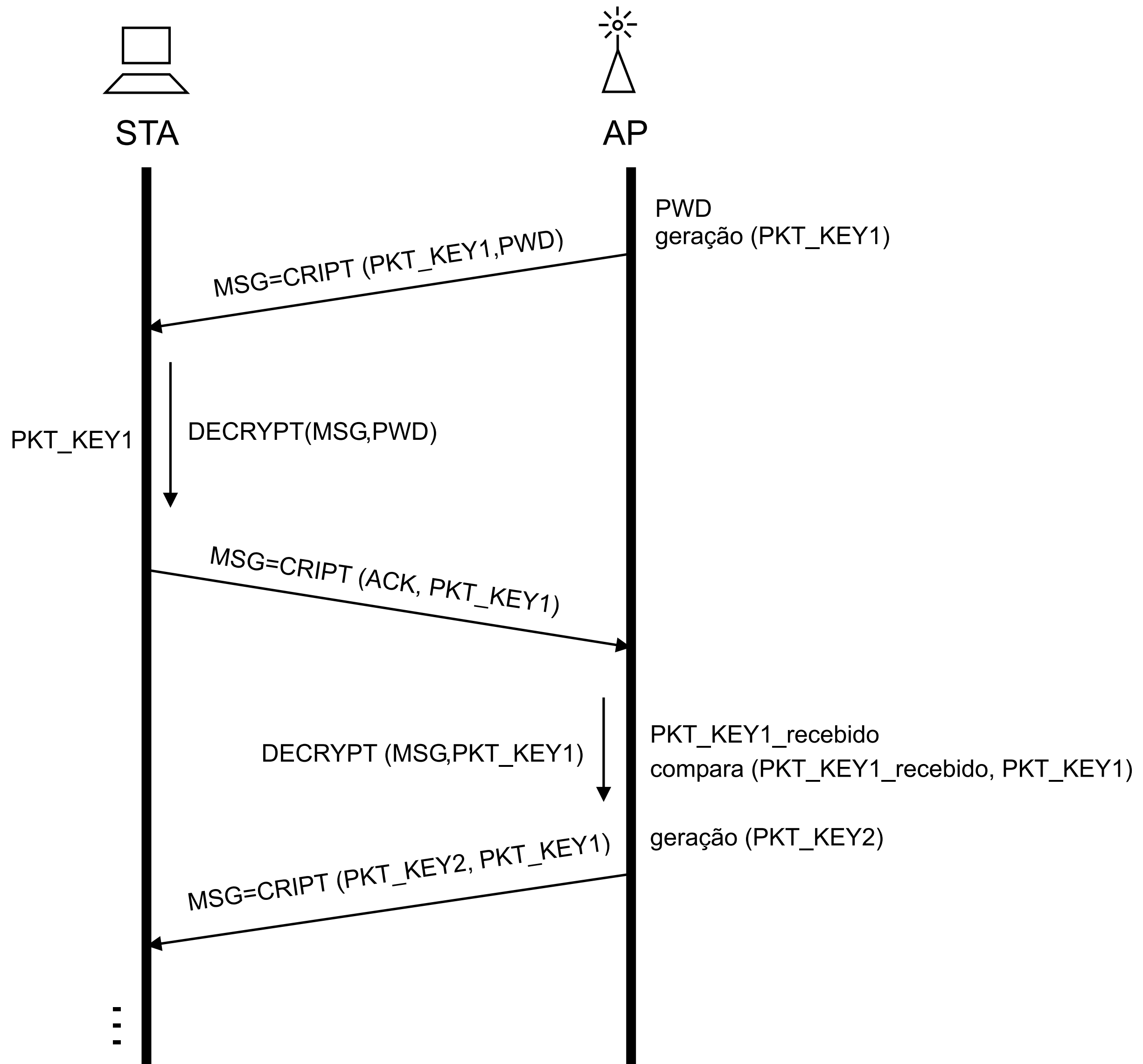

Figura 5.2: Troca típica de mensagens do sistema isAlive.

Neste ponto, a STA utiliza o conteúdo decifrado da mensagem recebida como a nova chave para encriptar uma mensagem de ACK e envia-la ao AP. Uma vez que o AP recebe esta mensagem, ele irá decripta-la. Caso esta seja decifrada com sucesso, fica provado que e STA em questão ainda está ativa na rede. Este processo se repete periodicamente até que a STA não mais responda as requisições do AP ou responda de forma incorreta (onde o AP não será capaz de decifrar a mensagem de ACK). Caso o AP não consiga estabelecer conexão com a STA ou não consiga decifrar corretamente a mensagem de ACK, a STA é considerada inativa. Uma vez que uma STA é considerada inativa, as regras do *firewall* no AP são editadas, de forma a retirar as permissões de acesso do IP que a STA em questão utilizara.

Estes procedimentos garantem a autenticidade da estação e evitam ataques replay e roubo de sessão, garantindo também, que o usuário da estação sem fio esteja enviando dados para o AP correto, evitando que um inimigo se passe pelo ponto de acesso.

5.3 Implementação do isAlive

Nesta seção, detalhes da implementação serão abordados, onde o ambiente de desenvolvimento do sistema isAlive será apresentado.

5.3.1 Aplicação Cliente Servidor

A proposta do isAlive é a instalação de um software (cliente da aplicação) na STA que responda a uma requisição de um *daemon*[2] (servidor da aplicação), instalado no AP (onde está presente o *firewall*). Este processo seria semelhante ao sistema de *keepalive*, descrito na seção 4.2.4. No entanto neste caso, informações adicionais seriam trocadas de forma segura entre clientes e servidor, com o objetivo de garantir tanto a autenticidade da estação, quanto a do AP. Isso evitaria que uma STA se associasse a um AP forjado por um atacante e não ao AP de produção, ao mesmo tempo que permitiria determinar que a STA requisitando conexão é realmente quem diz ser. Chamamos este procedimento de autenticação mútua.

[2]Um daemon é um programa que é executado em segundo plano e não possui terminal associado.

Nesta proposta o servidor da aplicação, chamado aqui de isAliveDaemon, estaria instalado no AP, de onde enviaria requisições de prova de vida periodicamente ao cliente da aplicação, chamado de isAliveStation, presente nas estações que estivessem autenticadas. O isAliveStation deve responder corretamente às requisições do isAliveDaemon para que seja feita a manutenção das respectivas conexões. A troca de mensagens realizada entre os clientes nas STAs e o servidor no AP está descrita em detalhes na seção 5.2.

5.3.1.1 Biblioteca TCPLIB

Um sub-produto deste projeto foi a criação e implementação de uma biblioteca, escrita em C++, cujo objetivo é facilitar o trabalho de programação para rede e criação de aplicações cliente-servidor.

Esta biblioteca, chamada TCPLIB, foi escrita sob o paradigma da orientação-a-objeto. Tal paradigma foi adotado, pois com ele é possível executar as tarefas de criação e inicialização de um servidor, assim como a conexão entre cliente-servidor, apenas com a instância de um objeto e chamando-se dois métodos do mesmo. O código fonte da biblioteca está em anexo a este documento.

5.3.2 Ambiente de Desenvolvimento

O isAlive é uma implementação em C/C++ do método de sondagem ativa descrito na seção 5.2, para verificar o desligamento de uma STA do AP. O livro [19] é uma das melhores referências quando se deseja escrever aplicações cliente-servidor. Esta linguagem de programação foi escolhida devido a sua portabilidade. Esta característica foi uma grande preocupação durante a implementação, uma vez que o programa cliente da aplicação isAlive, o isAliveStation, deveria ser capaz de executar da mesma forma em SOs (sistemas operacionais) diferentes. Os SOs alvos desta implementação do isAlive são Linux e Windows. Os SOs relacionados a seguir foram testados e são suportados pela atual implementação do isAlive. No entanto, qualquer outra versão dos sistemas Linux e Unix devem suportar adequadamente a aplicação cliente do isAlive, desde que instalados os devidos softwares, pré-requisitos para o funcionamento da mesma. Estes softwares e suas funcionalidades estão descritas a

seguir nesta seção.

- **isAliveDaemon**

    OpenBSD 3.3

    OpenBSD 3.4

- **isAliveStation**

    Windows XP/2000/ME/98.

    Slackware Linux 8.0 e superior.

    Red Hat Linux 7.0 ou superior.

    Conectiva Linux 6.0 ou superior.

    Aurora Linux 6.0 ou superior.

Para garantir esta característica de portabilidade à aplicação, foi utiliza a biblioteca Cygwin [20]. Este software emula um ambiente Linux dentro do Windows e é constituido de duas partes:

- Uma DLL (cygwin1.dll) que atua como uma camada que emula um sistema Linux, dando ao ambiente de desenvolvimento funcionalidades substanciais de uma API Linux.
- Uma coleção de ferramentas que fornecem aparência e funcionalidades Linux.

Com este software é possivel garantir a portabilidade do código em C/C++. Tanto em ambiente Cygwin sob Windows, quanto no próprio Linux, alguns softwares e bibliotecas são necessários e foram utilizados nesta implementação. Um desses softwares é o compilador gcc/g++ utilizado em ambos os ambientes. As principais bibliotecas e APIs utilizadas na implementação do isAlive foram a OpenSSL, citada na seção 5.3.3, e MYSQL++ citada na seção 5.3.4. A instalação de todas as bibliotecas e programas necessários ao funcionamento do isAlive serão vistos no capítulo 6.

5.3.3 Criptografia no isAlive

A biblioteca OpenSSL [8] foi utilizada para suporte ao desenvolvimento do sistema de criptografia das mensagens transmitidas entre clientes e servidor e para a troca de chaves entre STAs e AP.

O projeto OpenSSL é um esforço colaborativo para desenvolver um conjunto robusto de ferramentas, completo e de código aberto para implementação dos protocolos *Secure Sockets Layer* (SSL v2/v3) e *Transport Layer Security* (TLS v1), além de possuir uma biblioteca criptográfica de propósito geral. O projeto é gerenciado por uma comunidade de voluntários de todo o mundo que utilizam a Internet para se comunicar, planejar e desenvolver as ferramentas OpenSSL e sua documentação.

Dentro deste conjunto de ferramentas de criptografia oferecidas pelo OpenSSL, está a biblioteca EVP. Esta biblioteca fornece uma interface de alto nível para funções de criptografia. Foram utilizadas funções do tipo *EVP_Encrypt...*, disponíveis dentro da EVP, para criptografia simétrica.

O algoritmo de criptografia *Blowfish* foi utilizado nesta implementação da versão 1.0 do isAlive, em modo *Cipher Block Chaining* (CBC). Para isso foi utilizada a função *EVP_bf_cbc(void)*. A escolha do algoritmo desejado pode ser facilmente alterada em outras implementações. Para isso, basta apenas utilizar outra função EVP de criptografia simétrica.

5.3.3.1 Geração de Chaves

Atualmente, segurança de sistemas baseia-se em algoritmos de criptografia cada vez mais eficientes de forma a evitar ataques do tipo de identificação de padrões. No entanto, a segurança destes sistemas está intimamente relacionada à qualidade da geração de quantidades secretas de informação na constituição de senhas e chaves para criptografia.

A utilização de processos pseudo-aleatórios na geração destas quantidades secretas pode significar um risco para o sistema, uma vez que um atacante dispondo de tempo e recursos, pode reproduzir o ambiente onde as informações aleatórias secretas foram geradas. Isso proporcionaria uma redução do número de possibilidades a serem testadas, tornando o trabalho do atacante mais eficiente no sentido

de descobrir informações secretas.

Um verdadeiro gerador de números aleatórios requer uma fonte natural de aleatoriedade. Desenvolver dispositivos de hardware ou programas de software capazes de explorar fontes de aleatoriedade para gerar seqüências randômicas, não tendenciosas e livres de correlação é uma tarefa difícil. Segundo [21], página 5,*"Random numbers should not be generated with a method chosen at random"*. Alem disso, para aplicações de criptografia, o gerador em questão não pode ser facilmente submetido a observações ou manipulações por parte de um adversário. A RFC1750 [22] traz uma série de recomendações pertinentes a escolha de geradores de seqüências aleatórias. Em um trabalho sobre este tema [23], o autor realiza um estudo sobre a natureza dos diferentes tipos de geradores conhecidos e realiza ainda, uma avaliação dos mesmos, baseados em métodos descritos na literatura.

Assim, baseado nas referências citadas até agora e no que foi exposto, foi escolhida uma função de geração de seqüências aleatórias da biblioteca OpenSSL. A função utilizada é a $RAND_byte()$. Algum tratamento com relação ao sucesso na geração da aleatoriedade foi feito, de forma que se a função do OpenSSL não fosse capaz de gerar seqüências satizfatórias, novas seqüências deveriam ser geradas até que este objetivo fosse alcançado.

5.3.4 Acesso ao Banco de Dados

O servidor da aplicação necessita conectar-se ao banco de dados de autenticação e verificar que estações estão autenticadas e utilizando a rede. O acesso ao banco de dados MYSQL, pelo isAliveDaemon, é feito através da biblioteca MYSQL++ [24], que é uma API própria para acesso a este BD. O uso de tal interface de programação facilitou muito o trabalho de conectar, gravar e extrair informações do danco de dados.

5.4 Funcionamento do isAliveDaemon

Quando iniciado, o isAliveDaemon lê o arquivo isAliveDaemon.conf para obter os seguintes parametros:

- `DB_NAME`

  Nome do banco de dados.

- `DB_HOST`

  Endereço IP da máquina onde o banco de dados esta instalado.

- `DB_USER`

  Login para o banco de dados.

- `DB_PASSWD`

  Senha do banco de dados.

- `DB_TABLE_NAME`

  Nome da tabela do banco de dados dos estará a lista de ips das STAs que fazem parte do sistema.

A seguir pode ser visto um exemplo de um arquivo de configuração do isAliveDaemon.

```
**** Conteudo do Arquivo isAliveDaemon.conf ****
DB_NAME:strikein
DB_HOST:10.10.0.84
DB_USER:root
DB_PASSWD:super123
DB_TABLE_NAME:ipsValidos
```

Após o procedimento de inicialização, onde o programa lê o arquivo de configuração, ele executa uma série de procedimentos pré-estabelecidos de forma a verificar o estado de conexão de STAs clientes utilizando o serviço de rede sem fio. O programa instalado no AP, permanece rodando em segundo plano e acessa o banco de dados MYSQL (que pode estar no próprio AP ou em outra máquina) em busca de uma lista dos endereços IP das máquinas autenticadas através do StrikeIN. Além dos endereços, o programa procura pelas senhas dos usuários que se autenticaram em suas respectivas STAs.

De posse dos endereços das STAs e das senhas dos usuários, o *daemon* da aplicação isAlive segue tal lista e tenta se conectar aos clientes isAlive instalados

nas STAs. Quando o *daemon* se conecta a um dos clientes ele executa a operação descrita na seção 5.2. Este procedimento é feito para todas as estações na lista adquirida do banco de dados.

Em algum momento, a STA sendo sondada pode ser desligada ou sair do alcance do rádio do AP, fazendo com que o isAliveDaemon não consiga se conectar ao isAliveStation na STA. Isso irá fazer com que o isAliveDaemon, após N tentativas de conexão, retire o IP da STA considerada desativada do banco de dados e das permissões do *firewall*. A figura 5.3 ilustra o funcionamento do programa descrito acima para n estações registradas no banco de dados.

Mensagens de erro e mensagens importantes como a saída de estações (quando uma estação é considerada fora da rede e é então eliminada da lista de ips validos), são registradas nos arquivos de log do sistema. Isso é feito utilizando-se a função *syslog*() com seus devidos parametros. Em [19] é possivel achar a documentação da funcao *syslog*() e de outras função e *system calls* utilizadas nesta implementação.

5.5 Funcionamento do isAliveStation

O isAliveStation é um programa que deve ser inicializado antes da realização do login, na página de autenticação. Quando iniciado, o programa requisita a senha do usuário. Uma vez que o mesmo entra com sua senha correta, o isAliveStation torna-se um *daemon* e passa a ser executado em *background*. Foi escrita uma função chamada *daemonize*(), que realiza este procedimento de transformar o programa em *daemon*. Uma vez que isso acontece, o programa se "associa"a uma porta no sistema (atualmente a porta utilizada é 56789 e só pode ser alterada modificando-se o código fonte e recompilando-se o programa) e fica "ouvindo"essa porta, esperando uma conexão do isAliveDaemon no AP. Uma vez que o isAliveDaemon se conecta ao isAliveStation, a troca de mensagens descrita na figura 5.2 ocorre. Quando essa troca de mensagens termina, a conexão também é finalizada e o programa volta a esperar futuras conexões na mesma porta.

ISALIVEDAEMON

1⩾ n , i = 0

Geração de chave para
a i-ésima estação

1⩽ i ⩽n

i ++ ;

Consulta ao BD.
Lista de n IPs
e suas chaves
n>0

Probe i-ésima
STA

Editar regras
do firewall

Probe bem
sucedido

falha no probe
(falha de conexão)

Espera resposta
da STA

Acesso ao BD para apagar
registro do IP
da i-ésima STA

Resposta inválida

Resposta
correta

Acesso ao BD para
registro da chave

Figura 5.3: Esquema do funcionamento do isAliveDaemon.

Capítulo 6

Instalação e Configuração do isAlive

Neste capitulo, todo o processo de instalação e configuração do isAlive será apresentado.

6.1 Instalando o Banco de Dados MySQL

Nesta seção será descrita a instalação dos softwares cliente e servidor do banco de dados MYSQL.

6.1.1 Cliente MYSQL

- O AP deve ter instalado o cliente mySQL através dos seguintes passos:

  # Fazer o download do mySQL para o diretório */usr/src* e descompactar.

  $ cd /usr/src/mysql-version

  $ ./configure without-server

  $ gmake

  $ gmake install

- API C++ para mySQL Baixar e instalar a API para C++ para o mySQL (*www.mysql.com/downloads/api-mysql++.html*)

Fazer o download do mySQL++ para o diretório */usr/src* e descompactar.

$ automake

$ autoconf

$ ./config

$ gmake

$ gmake install

6.1.2 Servidor MYSQL

A primeira etapa do processo corresponderia a baixar o arquivo *mysql-VERSION.tar.gz* para o diretório */usr/src* e executar a série de comandos descrita a seguir.

$ groupadd mysql
$ useradd -g mysql mysql
$ tar -xvzf mysql-VERSION.tar.gz
$ cd mysql-VERSION
$ ./configure –prefix=/usr/local/mysql
$ gmake
$ gmake install
$ scripts/mysql_install_db
$ chown -R root /usr/local/mysql
$ chown -R mysql /usr/local/mysql/var
$ chgrp -R mysql /usr/local/mysql
$ cp support-files/my-medium.cnf /etc/my.cnf
$ /usr/local/mysql/bin/safe_mysqld–user=mysql

Deve-se configurar o MySQL de forma que ele seja invocado toda vez que o sistema for inicializado, para tanto devem-se acrescentar as seguintes linhas de comando no arquivo */etc/rc.local*:

```
if [ -x /usr/local/mysql/share/mysql/mysql.server ]; then
```

```
echo -n ' mysqld'; /usr/local/mysql/share/mysql/mysql.server start
fi
```

O MySQL permite que se incluam permissões de acessos utilizando-se de três restrições:

- Usuário
- Máquina (host)
- Banco de Dados

Desta forma estabelece-se que um usuário a partir de uma máquina tem acesso a um determinado banco de dados. Para tanto, foi criado um usuário denominado *strikein* com permissão de acesso da máquina AirStrike (AP) ao banco de dados *strikein*. Este banco de dados irá conter as informações pertinentes ao processo de autorização e autenticação dos usuários da rede sem fio local. Para criar este usuário, foram seguidos os seguintes passos:

```
$ mysql h localhost u root p
{Observe que a senha de root está em branco}
mysql > use mysql;
mysql > insert into user values(wstrike,wstrike,
PASSWORD(wstrike),Y,Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y,Y);
mysql > insert into db values(wstrike,strikein,
wstrike, Y, Y, Y, Y, Y, Y, Y, Y, Y, Y);
mysql > flush privileges;
```

Em anexo a este texto encontra-se o script utilizado para a criação do BD StrikeIN.

6.2 Instalação do isAliveDaemon

Os softwares necessários para o funcionamento do *daemon* são os seguintes:

- Banco de Dados MYSQL
- API C++ para MYSQL - MSQL++

- OpenSSL
- Código fonte do isAliveDaemon

Para instalar o isAlive no servidor, basta copiar o arquivo fonte do isAliveDaemon para uma pasta e executar o seguinte comando no Shell:

$ ./isAliveDaemon.sh

O arquivo isAliveDaemon.sh possui o seguinte conteúdo:

```
----------------------- Início do arquivo isAliveDaemon.sh ----------------------------------
echo "Compilando..." g++ -I/usr/local/include/ -I/usr/local/include/mysql -O2
-c isAliveDaemon.cpp tcp_lib.cpp

echo "Linkando..." g++ -g -O2 -L/usr/local/lib/mysql -o isAliveDaemon isAliveDaemon.o
-L/usr/local/lib/ -lsqlplus -lz -lmysqlclient -lz -lmysqlclient tcp_lib.cpp -lmcrypt
-lltdl libcrypto.a

echo "Terminando..." rm *.o
------------------------ Fim do arquivo isAliveDaemon.sh --------------------------------------
```

Para configurar o AP para iniciar o *daemon* isAlive no *boot* a seguinte linha deve ser incluida no arquivo */etc/rc.local*. Neste exemplo o executável do isAliveDaemon está localizado no diretório */usr/local/bin/*.

echo -n ’isAliveDaemon’; /usr/local/bin/isAliveDaemon   */dev/null*

É necessario também fazer as mudanças necessárias no arquivo de configuração do isaliveDaemon, isAliveDaemon.conf. Este arquivo deve estar no mesmo diretório do arquivo executavel do programa.

6.3 Instalação do isAliveStation

Para instalar o isAliveStation nos clientes que utilizão o sistema operacional Windows, os seguintes arquivos devem ser copiados para uma mesma pasta na máquina cliente (STA):

- isAliveStation.exe
- cygcrypto-0.9.7.dll

- Cygwin1.dll

Para instalar o isAliveStation em clientes Linux, basta copiar os arquivos fonte do isAliveStation para uma pasta e executar o seguinte comando no Shell:

$ ./isAliveStation.sh

O arquivo isAliveStation.sh possui o seguinte conteúdo:

```
---------------------- Início do arquivo isAliveStation.sh -----------------------------------
echo "Compilando e linkando..." g++ isAliveStation.cpp tcp_lib.cpp -o
isAliveStation -lcrypto -Wno-deprecated

echo "Terminando..." rm *.o
----------------------- Fim do arquivo isAliveStation.sh -------------------------------------
```

Capítulo 7

Conclusões e Trabalhos Futuros

Com este trabalho, foi dado um passo importante na direção de resolver um problema de segurança bem específico em redes sem fio, envolvendo desligamento de estações. Um protocolo de detecção de desligamento de estação DPD (*Dead Peer Detection*) foi idealizado um aplicativo cliente-servidor que implementa este protocolo foi escrito. O protocolo e a aplicação recebem o nome de isAlive. Este aplicativo é capaz de identificar quando uma estação sem fio se desliga da rede por qualquer motivo e pode alterar configurações e regras de *firewall* do sistema. Este software detecta e fecha portas abertas em *firewalls* baseado no estado das conexões das estações sem fio que fazem partes da rede evitando ataques de replay e ipspoofing.

O ambiente de desenvolvimento de aplicativo para redes de computadores foi elaborado de forma a ser ampliado de modo simples e efetivo, caso novas funcionalidades sejam requeridas. Estas podem ser facilmente incluídas, devido ao alto grau de encapsulamento do código, permitindo assim a constante evolução deste trabalho. Trata-se de um subproduto deste trabalho chamado aqui de TCPLIB, uma biblioteca, escrita em C++, cujo objetivo é facilitar o trabalho de programação para rede e criação de aplicações cliente-servidor. Esta linguagem de programação foi escolhida devido a sua portabilidade. Esta característica foi uma grande preocupação durante a implementação, uma vez que o programa cliente da aplicação, o isAliveStation, deveria ser capaz de executar da mesma forma em sistemas operacionais diferentes.

Por fim, é válido dizer que este trabalho não deve ser visto como algo isolado, acabado. Da mesma forma que a tecnologia está sempre em desenvolvimento, cabe aos futuros desenvolvedores interessados em dar continuidade a este trabalho, implementar as novas tecnologias oferecidas, de forma a manter este sistema sempre atualizado e permitindo seu uso contínuo por um longo tempo. Algumas propostas de trabalhos futuros podem ser citadas:

- O método de sondagem ativa das estações pode ser feito em paralelo e não em sistema de polling. Uma avaliação comparativa deve ser feita das implementações com sondagem em série e em paralelo. Apenas a sondagem em série foi implementada.

- O mecanismo de chave pré-compartilhada, utilizando-se a princípio a senha do usuário, foi uma alternativa simples e eficaz para realizar a criptografia e a autenticação do protocolo proposto. Alternativas a este método podem ser implementadas, como por exemplo o uso de certificados digitais. Isso poderia minimizar os problemas relacionados a possiveis ataques, como ataques de dicionário, por exemplo.

Apêndice A

Código Fonte da Biblioteca TCP para Programação para Rede

```
// tcp_lib.cpp (v1.0) -
// Bruno Astuto Arouche Nunes
// 2002/2
// 28-11-2002
//---------------------------------------------------------------------------
#include <sys/types.h>
#include <sys/time.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include "tcp_lib.h"

#define DEBUG 0
#define BACKLOG 5  // max number of clients
#define BACKLOGSINGLE 1  // max number of clients

char *tcpipErrorMsg[] =
{
"OK",
"TCP/IP     : ERROR IN gethostbyname FUNCTION ...\n",
"TCP/IP     : ERROR IN socket FUNCTION ...\n",
"TCP/IP     : ERROR IN connect FUNCTION ...\n",
```

```
"TCP/IP     : ERROR IN close FUNCTION ...\n",
"TCP/IP     : ERROR IN recv FUNCTION ...\n",
"TCP/IP     : ERROR IN send FUNCTION ...\n",
"TCP/IP     : ERROR IN bind FUNCTION ...\n",
"TCP/IP     : ERROR IN listen FUNCTION ...\n",
"TCP/IP     : ERROR IN setsockopt FUNCTION ...\n",
"TCP/IP     : ERROR IN accept FUNCTION ...\n",
"TCP/IP     : ERROR IN gethostname FUNCTION ...\n",
"TCP/IP     : ERROR IN getpeername FUNCTION ...\n"
};

/****************************************************/
/* tcpIp class functions */
/****************************************************/

// constructor of the tcpIp class
tcpIp::tcpIp()
{
#if DEBUG
cout << "DEBUG: IN THE CONSTRUCTOR OF tcpIp::tcpIp()!" << endl;
#endif DEBUG
constructorError = OK;
nameOfTheHost = 0;
ipOfTheHost = 0;
}

// destructor of the tcpIp class
tcpIp::~tcpIp()
{
#if DEBUG
cout << "DEBUG: IN THE DESTRUCTOR OF tcpIp::tcpIp()!" << endl;
#endif DEBUG

if (nameOfTheHost) delete (nameOfTheHost);
if (ipOfTheHost) delete (ipOfTheHost);
}

void tcpIp::setNameOfTheHost(struct hostent *h)
```

```
{
// initialize the nameOfTheHost atribute
nameOfTheHost = new char[strlen(h->h_name) + 1];
strcpy(nameOfTheHost, h->h_name);
}

char *tcpIp::getNameOfTheHost(int print = 0)
{
// return the nameOfTheHost atribute
if (print)
{
cout << nameOfTheHost << endl;
}
return(nameOfTheHost);
}

void tcpIp::setIpOfTheHost(struct hostent *h)
{
// initialize the ipOfTheHost atribute
ipOfTheHost = new char[strlen( inet_ntoa(*((struct in_addr *)h->h_addr)) ) + 1];
strcpy(ipOfTheHost, inet_ntoa(*((struct in_addr *)h->h_addr)) );
}

char *tcpIp::getIpOfTheHost(int print = 0)
{
// return the ipOfTheHost atribute
if (print)
{
cout << ipOfTheHost << endl;
}
return(ipOfTheHost);
}

ulong tcpIp::simpleSendData(char *buffer)
{
ulong retcode = OK;
retcode = simpleSendData((byte*)buffer);
```

```
return(retcode);
}
ulong tcpIp::sendData(char *buffer , int buflen)
{
ulong retcode = OK;
retcode = sendData((byte*)buffer, buflen);
return(retcode);
}

ulong tcpIp::simpleSendData(byte *buffer)
{
int bytes_sent = 0; // how many bytes we´ve sent at that moment
ulong buffer_length = 0;

buffer_length = strlen((char*)buffer)+1;

#if DEBUG
cout << "DEBUG: IN simpleSendData FUNCTION 0 \n" << endl;
#endif

// txmsg = 0;
/*
if(txmsg) //safe delete
{
delete(txmsg);
txmsg = 0;
}

txmsg = new byte[buffer_length];
*/
#if DEBUG
cout << "DEBUG: IN simpleSendData FUNCTION 1 \n" << endl;
#endif

#ifdef WIN32
/*ulong i = 0;
while( i < strlen(buffer) )
{
```

```
txmsg[i] = '\0';
i++;
}*/
memset(txmsg, '\0', 2048);
#else
bzero(txmsg, 2048);
#endif

//bcopy(txmsg, buffer, buffer_length);
memcpy(txmsg, buffer, buffer_length);
//memccpy(txmsg, buffer, '\0', buffer_length);
//strcpy(txmsg, buffer);

#if DEBUG
cout << "DEBUG: txmsg = " << txmsg << endl;
#endif

#if DEBUG
cout << "DEBUG: IN simpleSendData FUNCTION 2 \n" << endl;
//cout << "DEBUG: total_length = "<< total_length << " \n" << endl;
#endif

#ifdef WIN32
bytes_sent = send(socketDescriptor, (char*)txmsg, strlen((char*)txmsg), 0);
if (bytes_sent < 0) { return(ERROR_SEND); }
#else
bytes_sent = send(socketDescriptor, txmsg, strlen((char*)txmsg), 0);
if (bytes_sent < 0) { return(ERROR_SEND); }
#endif

#if DEBUG
cout << "txmsg ->" << txmsg << "<-" << endl;
#endif

return(OK);
}

ulong tcpIp::sendData(byte *buffer, int buflen)
```

```
{
ulong total = 0; // how many bytes we´ve sent
ulong bytes_left = 0; // how many we have left to send
ulong total_length = 0;
int bytes_sent = 0; // how many bytes we´ve sent at that moment
ulong buffer_length = 0;

buffer_length = buflen;

#if DEBUG
cout << "DEBUG: IN sendData FUNCTION 0 \n" << endl;
#endif

#if DEBUG
cout << "DEBUG: IN sendData FUNCTION 1 \n" << endl;
#endif

#ifdef WIN32
memset(txmsg, '\0', DATA_SIZE);
#else
bzero(txmsg, DATA_SIZE);
#endif

memcpy(txmsg, buffer, buffer_length);

#if DEBUG
cout << "DEBUG: txmsg = " << txmsg << endl;
#endif

bytes_left = buflen;
total_length = buflen;//strlen((char*)txmsg);

#if DEBUG
cout << "DEBUG: IN sendData FUNCTION 2 \n" << endl;
cout << "DEBUG: total_length = "<< total_length << " \n" << endl;
#endif

while(total < total_length)
```

```
{
bytes_sent = send(socketDescriptor, (char*)(txmsg+total), bytes_left, 0);
if (bytes_sent < 0) { return(ERROR_SEND); }
total += bytes_sent;
bytes_left -= bytes_sent;
}

#if DEBUG
cout << "txmsg ->" << txmsg << "<-" << endl;
#endif

return(OK);
}

ulong tcpIp::receiveData(ulong maxlength, int *bytes_recved)
{
#if DEBUG
cout << "DEBUG: IN reciveTcpData FUNCTION \n" << endl ;
#endif DEBUG

bytes_recved = 0;

#ifdef WIN32
memset(rxmsg, '\0', DATA_SIZE);
#else
bzero(rxmsg, DATA_SIZE);
#endif

#if DEBUG
cout << "DEBUG: IN recv FUNCTION \n" << endl;
cout << &rxmsg << endl;
#endif

#ifdef WIN32
if( ( *bytes_recved = recv(socketDescriptor, (char*)rxmsg, DATA_SIZE, 0) ) < 0 )
return(ERROR_REVC);
#else
if( ( *bytes_recved = recv(socketDescriptor, rxmsg, DATA_SIZE, 0) ) < 0 )
```

```
return(ERROR_REVC);
#endif
#if DEBUG
cout << "DEBUG: IN recv FUNCTION 2 \n" << endl;
cout << "DEBUG: bytes_recved = "<< bytes_recved << " \n" << endl;
#endif

#if DEBUG
cout << "rxmsg ->" << rxmsg << "<-" << endl;
#endif DEBUG

    return(OK);
}

byte *tcpIp::getRXmsg(int print = 0)
{
// return the rxmsg atribute
if (print)
{
cout << rxmsg << endl;
}
return(rxmsg);
}

byte *tcpIp::getTXmsg(int print = 0)
{
// return the txmsg atribute
if (print)
{
cout << txmsg << endl;
}
return(txmsg);
}

ulong tcpIp::getConstructorError()
{
return(constructorError);
}
```

```
#ifdef WIN32
void tcpIp::callWSAStartup()
{
// WSAStartup call -- Needed for windows sockets
WORD wVersionRequested;
WSADATA wsaData;
int err;

wVersionRequested = MAKEWORD( 2, 2 );

err = WSAStartup( wVersionRequested, &wsaData );
if ( err != 0 )
{
/* Tell the user that we could not find a usable */
/* WinSock DLL.                                  */
return;
}

/* Confirm that the WinSock DLL supports 2.2.  */
/* Note that if the DLL supports versions greater    */
/* than 2.2 in addition to 2.2, it will still return */
/* 2.2 in wVersion since that is the version we      */
/* requested.                                        */

if ( LOBYTE( wsaData.wVersion ) != 2 ||
HIBYTE( wsaData.wVersion ) != 2 )
{
/* Tell the user that we could not find a usable */
/* WinSock DLL.                                  */
WSACleanup( );
return;
}
/* The WinSock DLL is acceptable. Proceed. */
}
#endif

// END OF tcp_client class functions =================================
```

```
/*******************************************************************/
/* tcpClient class functions   */
/*******************************************************************/

// constructor
tcpClient::tcpClient(char *serverAddr, char *p, int v = 0)
{
int myport = atoi(p);
int retcode = OK;

if( (retcode = setTcpClient(serverAddr, myport, v)) )
{
constructorError = retcode;
}
}

// constructor
tcpClient::tcpClient(char *serverAddr, ushort p, int v = 0)
{
int retcode = OK;

if( (retcode = setTcpClient(serverAddr, p, v)) )
{
constructorError = retcode;
}
}

// Called by the constructor to initialize parameters for the client.
ulong tcpClient::setTcpClient(char *serverAddr, ushort p, int v)
{

#ifdef WIN32
callWSAStartup();
#endif

#if DEBUG
cout << "DEBUG: IN tcpClient CONSTRUCTOR \n" << endl;
```

```
#endif

struct hostent *h; //Hold the server information.
constructorError = OK;
serverAddress = 0;

// initialize the serverAddress atribute
serverAddress = new char[strlen(serverAddr) + 1];
strcpy(serverAddress, serverAddr);
// initialize the port atribute
port = p;
// initialize the verbose atribute
verbose = v;

#if DEBUG
cout << "DEBUG: serverAddress = "<< serverAddress << " \n" << endl;
#endif

if( ( h = gethostbyname(serverAddress) ) == NULL ) // get the host info
{
if (verbose)
{
cout << "DEBUG: constructorError = " << constructorError << endl;
}
return(ERROR_GETHOSTBYNAME);
}
else
{
// initialize the nameOfTheHost atribute
setNameOfTheHost(h);
// initialize the ipOfTheHost atribute
setIpOfTheHost(h);

// Setting structure parameters ************************
dest_addr.sin_family = SOCK_FAMILY;
dest_addr.sin_port   = htons(port);
dest_addr.sin_addr   = *((struct in_addr *)h->h_addr);
memset(&(dest_addr.sin_zero), '\0', 8); // zero to the rest of the structure
```

```
// Setting structure parameters *******END**************
}

#if DEBUG
cout << "DEBUG: LEAVING tcpClient CONSTRUCTOR! \n" << endl;
#endif

return(OK);
}

ulong tcpClient::initializeClientConnection()
{
        fd_set fds;
        int n, timeout = 5;
        struct timeval tv;

        #if DEBUG
cout << "DEBUG: IN initializeClientConnection FUNCTION \n" << endl;
#endif

    // Opening socket ************************
if ( ( socketDescriptor = socket(SOCK_FAMILY, SOCK_TYPE, SOCK_PROTOCOL) ) < 0 )
return(ERROR_OPEN_SOCKET);
if (fcntl (socketDescriptor, F_SETFL, O_NONBLOCK) < 0) {
                printf("erro no fcntl\n");
                exit(1);
              }
if (verbose)
{
cout << "TCP/IP client    : Socket Opened ... " << endl;
}

 // set up the struct timevalfor the timeout
         tv.tv_sec = timeout;
         tv.tv_usec = 0;

    // Connecting to server *******************
if (verbose)
```

```
{
cout << "Trying to connect...." << endl;
}

 while(1) {
    if ( (connect(socketDescriptor, (struct sockaddr *) &dest_addr, sizeof(sockaddr_in)) < 0) && (err
           if ((errno != EINTR) && (errno != EINPROGRESS)) {

if ((errno == ECONNREFUSED)||(errno == EINVAL)) {
if (verbose) fprintf(stderr, "Connection Refused!\n");
return (ERROR_OPEN_SOCKET);
}

if (errno == EALREADY) {
   if (verbose) fprintf(stderr, "Connection timed out!\n");
   return (ERROR_OPEN_SOCKET);
}
                                perror("connect");
                                return(ERROR_OPEN_SOCKET);
}
    } else {
                       if (verbose)
                       {
                            cout << "TCP/IP client    : Connected ...\n";
                            cout << "Host Name (IP)   : " << nameOfTheHost << " (" << ipOfTheHost <<
                       }
                       if (fcntl (socketDescriptor, F_SETFL, O_SYNC) < 0) {
if (verbose)
printf("erro no fcntl\n");
exit(1);
                       }

if (verbose) printf ("Connected succesfully!\n");

return (OK);
    }

        FD_ZERO(&fds);
```

```
        FD_SET(socketDescriptor,&fds);

        while (select (getdtablesize (), NULL, &fds, NULL, &tv) < 0) {
           if (errno != EINTR) {
perror ("select");
return(1);
}
}
 }
return(OK);
}

ulong tcpClient::endClient()
{
int i = 0;

#if DEBUG
cout << "DEBUG: IN endClient FUNCTION \n" << endl;
#endif

#ifdef WIN32
// close socket for Win32
WSACleanup();
return(OK);
#else
// close socket for unix
if( (i = close(socketDescriptor)) )
{
return(ERROR_CLOSE);
}
#endif

return(OK);
}

// destructor
tcpClient::~tcpClient()
{
```

```
#if DEBUG
cout << "DEBUG: IN DESTRUCTOR ~tcpClient() \n" << endl;
#endif

if(serverAddress) delete(serverAddress);

// END tcp client
if ( (retCode = endClient()) )
{
cout << tcpipErrorMsg [retCode] << endl;
}
}
// END OF tcp_client class functions ================================


/**************************************************/
/* tcpServer class functions */
/**************************************************/

// constructor
tcpServer::tcpServer(char *p, int v = 0)
{
int myport = atoi(p);
int retcode = OK;

if( (retcode = setTcpServer(myport, v)) )
{
constructorError = retcode;
}
}

// constructor
tcpServer::tcpServer(ushort p, int v)
{
int retcode = OK;

if( (retcode = setTcpServer(p, v)) )
{
```

```
constructorError = retcode;
}
}

ulong tcpServer::setTcpServer(ushort p, int v)
{
#ifdef WIN32
callWSAStartup();
#endif

#if DEBUG
cout << "DEBUG: IN tcpServer CONSTRUCTOR \n" << endl;
#endif

struct hostent *h; //Hold the server information.
constructorError = OK;
ipOfThePeer = 0; //Hold connector's IP address
nameOfThePeer = 0; //Hold connector's name
portOfThePeer = 0; //Hold connector's connection port

// initialize the serverAddress atribute
char serverAddress[DATA_SIZE];
if( ( gethostname(serverAddress, sizeof(serverAddress)) ) < 0 ) // get the host name
{
if (verbose)
{
constructorError = ERROR_GETHOSTNAME;
cout << "1: tcpServer Constructor Error = " << constructorError << endl;
}
return(ERROR_GETHOSTNAME);
}

// initialize the port atribute
port = p;

// initialize the verbose atribute
verbose = v;
```

```
if( ( h = gethostbyname(serverAddress) ) == NULL ) // get the host info
{
if (verbose)
{
constructorError = ERROR_GETHOSTBYNAME;
cout << "2: tcpServer Constructor Error = " << constructorError << endl;
}
return(ERROR_GETHOSTBYNAME);
}
else
{
// initialize the nameOfTheHost atribute
setNameOfTheHost(h);
// initialize the ipOfTheHost atribute
setIpOfTheHost(h);

// Setting structure parameters ************************
server_addr.sin_family = SOCK_FAMILY;
server_addr.sin_port   = htons(port);
server_addr.sin_addr   = *((struct in_addr *)h->h_addr);
memset(&(server_addr.sin_zero), '\0', 8); // zero to the rest of the structure
// Setting structure parameters *******END**************
}

#if DEBUG
cout << "DEBUG: LEAVING tcpServer CONSTRUCTOR! \n" << endl;
#endif

return(OK);
}

ulong tcpServer::initializeSingleServer()
{
#ifdef WIN32
int sin_size;
#else
socklen_t sin_size;
#endif
```

```
retCode = initializeServer();

sin_size = sizeof(struct sockaddr_in);
if ( ( socketDescriptor = accept(sock_fd, (struct sockaddr *) &client_addr, &sin_size) ) < 0 )
{
#ifdef WIN32
// close socket for Win32
WSACleanup();
return(ERROR_ACCEPT);
#else
// close socket for unix
close(sock_fd);
return(ERROR_ACCEPT);
#endif
}

if (verbose)
{
cout << "TCP/IP server      : Connection accepted from " << inet_ntoa(client_addr.sin_addr) << endl;
}

#ifdef WIN32
#else
if ( close(sock_fd) )
{
return(ERROR_CLOSE);
}
#endif

return(retCode);
}

ulong tcpServer::initializeServer()
{
#ifdef WIN32
char yes = 1;
#else
```

```
int  yes = 1;
#endif

// Opening socket ************************
if ((sock_fd = socket(SOCK_FAMILY, SOCK_TYPE, SOCK_PROTOCOL)) < 0)
return(ERROR_OPEN_SOCKET);

if (verbose)
{
cout << "TCP/IP server      : Socket Opened ..." << endl;
}

// Forcing the port to be released ********
// Lose the pesky "Address alredy in use" error message *********
setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));

// Binding to port *******************
if ( ( bind( sock_fd, (struct sockaddr *) &server_addr, sizeof(server_addr) ) ) < 0 )
{
#ifdef WIN32
// close socket for Win32
WSACleanup();
return(ERROR_BIND);
#else
// close socket for unix
close(sock_fd);
return(ERROR_BIND);
#endif
}

if (verbose)
{
cout << "TCP/IP server      : Bind to port " << port << " ..." << endl;
}

// Listenig to port ****************************************************
// this server accept one single connection. (BACKLOGSINGLE = 1)
if ((listen(sock_fd, BACKLOGSINGLE)) < 0)
```

```
{
#ifdef WIN32
// close socket for Win32
WSACleanup();
return(ERROR_LISTEN);
#else
// close socket for unix
close(sock_fd);
return(ERROR_LISTEN);
#endif
}

if (verbose)
{
cout << "TCP/IP server      : Listening port " << port << " ..." << endl;
cout << "Server Name (IP)   : " << nameOfTheHost << " (" << ipOfTheHost << ")" << endl;
}

return(OK);
}

ulong tcpServer::setPeerInfo()
{
#if DEBUG
cout << "DEBUG: IN setPeerInfo() METHOD!!!" << endl;
#endif

int retcode = OK;
struct sockaddr_in peer_addr; //Hold connector's addr

#ifdef WIN32
int peer_addr_len;
#else
socklen_t peer_addr_len;
#endif

if( ( getpeername(socketDescriptor, (struct sockaddr *) &peer_addr, &peer_addr_len) ) < 0 )
{
```

```
return(ERROR_GETPEERNAME);
}

if ( (retcode = setPeerInfo(peer_addr)) )
{
return(retcode);
}

return(OK);
}

ulong tcpServer::setPeerInfo(struct sockaddr_in peer_addr)
{
struct hostent *h;

if( ( h = gethostbyname( inet_ntoa(peer_addr.sin_addr) ) ) == NULL ) // get the conectors info
{
return(ERROR_GETHOSTBYNAME);
}

#if DEBUG
cout << "ipOfThePeer::::::" << endl;
#endif
// initialize the ipOfThePeer atribute
ipOfThePeer = new char[strlen( inet_ntoa(*((struct in_addr *)h->h_addr)) ) + 1];
strcpy(ipOfThePeer, inet_ntoa(*((struct in_addr *)h->h_addr)) );

#if DEBUG
cout << "nameOfThePeer::::::" << endl;
#endif
// initialize the nameOfThePeer atribute
nameOfThePeer = new char[strlen(h->h_name) + 1];
strcpy(nameOfThePeer, h->h_name);

#if DEBUG
cout << "portOfThePeer::::::" << endl;
#endif
// initialize the portOfThePeer atribute
```

```
portOfThePeer = ntohs(peer_addr.sin_port);

return(OK);
}

char *tcpServer::getPeerAddr(int print = 0)
{
// return the ipOfThePeer atribute
if (print)
{
cout << ipOfThePeer << endl;
}
return(ipOfThePeer);
}

char *tcpServer::getPeerName(int print = 0)
{
// return the nameOfThePeer atribute
if (print)
{
cout << nameOfThePeer << endl;
}
return(nameOfThePeer);
}

int tcpServer::getPeerPort(int print = 0)
{
// return the portOfThePeer atribute
if (print)
{
cout << portOfThePeer << endl;
}
return(portOfThePeer);
}

ulong tcpServer::disconectClient()
{
int i = 0;
```

```
#if DEBUG
cout << "DEBUG: IN endServer FUNCTION \n" << endl;
#endif

#ifdef WIN32
// close socket for Win32
WSACleanup();
return(OK);
#else
// close socket for unix
if( (i = close(socketDescriptor)) )
{
return(ERROR_CLOSE);
}
#endif

return(OK);
}

// destructor
tcpServer::~tcpServer()
{
#if DEBUG
cout << "DEBUG: IN DESTRUCTOR ~tcpServer() \n" << endl;
#endif

if (nameOfThePeer) delete(nameOfThePeer);
if (ipOfThePeer) delete(ipOfThePeer);

#ifdef WIN32
// close socket for Win32
WSACleanup();
#endif
}
// END OF tcp_server class functions ==================================
```

```
// tcp_lib.h (v7.3)
// Bruno Astuto Arouche Nunes
// 2002/2
// 24-10-2002
//-----------------------------------------------------------------------

#ifndef _TCPLIB_H_
#define _TCPLIB_H_

#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#ifdef WIN32
// includes for windows
#include <winsock2.h>
#else
// includes for unix
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netdb.h>
#endif

#define OK 0
#define ERROR_GETHOSTBYNAME 1
#define ERROR_OPEN_SOCKET 2
#define ERROR_CONNECT 3
#define ERROR_CLOSE 4
#define ERROR_REVC 5
#define ERROR_SEND 6
#define ERROR_BIND 7
#define ERROR_LISTEN 8
#define ERROR_SETSOCKOPT 9  // not used
#define ERROR_ACCEPT 10
```

```
#define ERROR_GETHOSTNAME 11
#define ERROR_GETPEERNAME 12

#define SIZE_DATA 1024
#define DATA_SIZE 10240

#define SOCK_FAMILY AF_INET // socket family
#define SOCK_TYPE SOCK_STREAM // socket type
#define SOCK_PROTOCOL IPPROTO_TCP // used protocol

#ifdef WIN32
#define False false
#define True true
#else
#define FALSE false
#define False false
#define TRUE true
#define True true
#endif



#ifndef byte
#define byte unsigned char
#endif
#ifndef ulong
#define ulong unsigned long
#endif
#ifndef ushort
#define ushort unsigned short
#endif

extern char *tcpipErrorMsg[];

/* -------------- PROTOTYPES --------------- */

//////////////////////////////////////
//    tcpIp     //
//////////////////////////////////////
```

```
class tcpIp
{
protected: // atributes
char *nameOfTheHost;
char *ipOfTheHost;
byte rxmsg[DATA_SIZE];
byte txmsg[DATA_SIZE];
int retCode;
int verbose; // flag to dispay messages on screen.
int constructorError;
ushort port;

#ifdef WIN32
protected:
void callWSAStartup(); // method necessary to the use of WinSocks
#endif

public: // methods
tcpIp(); // constructor
~tcpIp(); // destructor
void setNameOfTheHost(struct hostent *h);
void setIpOfTheHost(struct hostent *h);
char *getNameOfTheHost(int print);
char *getIpOfTheHost(int print);
byte *getRXmsg(int print);
byte *getTXmsg(int print);
ulong simpleSendData(byte *buffer);
ulong sendData(byte *buffer, int buflen);
ulong simpleSendData(char *buffer);
ulong sendData(char *buffer, int buflen);
ulong receiveData(ulong maxlength, int *bytes_recved);
ulong getConstructorError();

int socketDescriptor;
};// end of tcpIp class


//////////////////////////////////////////
```

```
// tcpClient //
////////////////////////////////////////
class tcpClient : public tcpIp
{
private: // atributes
char *serverAddress;
struct sockaddr_in dest_addr; //Hold the destination addr.

private: // methods
ulong setTcpClient(char *serverAddr, ushort p, int v = 0);

public: // methods
tcpClient(char *serverAddr, ushort p, int v); // constructor
tcpClient(char *serverAddr, char *p, int v); // constructor
~tcpClient(); // destructor
ulong initializeClientConnection();
ulong endClient();

};// end of tcp_client class



////////////////////////////////////
//    tcpServer //
////////////////////////////////////
class tcpServer : public tcpIp
{
private: // atributes
char *ipOfThePeer; //Hold connector's IP address
char *nameOfThePeer; //Hold connector's name
int portOfThePeer; //Hold connector's connection port

private: // methods
ulong setTcpServer(ushort p, int v = 0);

public: // methods
tcpServer(ushort p, int v); // constructor
tcpServer(char *p, int v); // constructor
~tcpServer(); // destructor
```

```
ulong initializeSingleServer();
ulong initializeServer();
ulong disconectClient();
ulong setPeerInfo();
ulong setPeerInfo(struct sockaddr_in peer_addr);
char *getPeerAddr(int print);
char *getPeerName(int print);
int getPeerPort(int print);

struct sockaddr_in server_addr; //Hold server's addr (my addr)
struct sockaddr_in client_addr;  //Hold connector's information
int sock_fd;

};// end of tcp_server class

/* ----------- END OF PROTOTYPES ------------ */

#endif
```

Apêndice B

Código Fonte do isAliveStation

```
// isAliveStation.h (v1.0) *** For LINUX/UNIX and Windows98/2k/XP(with CYGWIN) ***
// Author: Bruno Astuto Arouche Nunes
// Date: 22-12-2003
// 2003/2
//
// g++ isAliveStation.cpp tcp_lib.cpp -o isAliveStation -lcrypto -Wno-deprecated
//------------------------------------------------------------------------------------------

#include <stdlib.h>
#include <fstream.h>
#include <openssl/evp.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <syslog.h>
#include <signal.h>
#include <pwd.h>
#include <unistd.h>
#include <stdio.h>
#include "tcp_lib.h"

#define DEBUG_STA 0
#define VERBOSE 1
#define KEY_SIZE 17
#define PORTNUMBER 56789
#define ACK_MSG "200 ACK"
```

```
#define MAX_FD 64


// ****** PROTOTYPES ****** //
ulong answerProbe(unsigned char *firstKey, int new_fd);
ulong initializeSingleTcpServer(ushort port, int *socketDescriptor, int verbose);
void daemonize(const char *pname, int facility);
void writeLog(char *log_msg);

static unsigned char key[KEY_SIZE];



// isAliveStation.cpp (v1.0) *** For LINUX/UNIX and Windows98/2k/XP (with CYGWIN) ***
// Author: Bruno Astuto Arouche Nunes
// Date: 22-12-2003
// 2003/2
//
// g++ isAliveStation.cpp tcp_lib.cpp -o isAliveStation -lcrypto -Wno-deprecated
//------------------------------------------------------------------------------------------

#include "isAliveStation.h"

#define LOG_FILE "isAliveStation.log"

int main(int argc, char **argv)
{
ulong retCode = OK;
int socketDescriptor = 0;
socklen_t sin_size; //    int sin_size = 0; //mudei aqui
int new_fd = 0, validPasswd;
struct sigaction sa;
struct sockaddr_in client_addr; //Hold the destination addr
char *p;
char buf[DATA_SIZE]; memset(buf, '\0', DATA_SIZE);
char configParamName[DATA_SIZE] = "\0"; memset(configParamName, '\0', DATA_SIZE);
char configLine[DATA_SIZE] = "\0"; memset(configLine, '\0', DATA_SIZE);
char type[KEY_SIZE] = "\0"; memset(type, '\0', KEY_SIZE);
```

```
char retype[KEY_SIZE] = "\0"; memset(retype, '\0', KEY_SIZE);

memset(key, '\0', KEY_SIZE);

//****************************************************************************************

// Set the variable DEBUG_STA in isAliveStation.h to 0 to disable DEBUG MODE,
// or to 1 to se the debug messages.
#if DEBUG_STA
cout << "*** DEBUG MODE ********************************* isAliveStation Started..." << endl;
#endif

#if !DEBUG_STA
cout << "************************************************ isAliveStation Started..." << endl;
#endif

validPasswd=0;
while ( ! validPasswd )
{
strncpy( type,   getpass("TYPE PASSWORD  :"), sizeof(type)-1 );
strncpy( retype, getpass("RETYPE PASSWORD:"), sizeof(retype)-1);

if ( strncmp(type, retype, sizeof(type) ) == 0)
{
validPasswd=1;
}
else
{
cout << "Invalid Password! Try again!" << endl;
writeLog("Invalid Password! Try again!");
}
}

strncat(retype, "0000000000000000", KEY_SIZE-strlen(retype)-1);
retype[sizeof(retype)-1]='\0';

strncpy((char*)key, retype, sizeof(key) );
```

```
#if DEBUG_STA
cout << "YOUR complete PASSWORD IS: (" << key << ")" << endl;
#endif
writeLog("isAliveStation Started...");
//****************************************************************************************

//Make the process a daemon.//
daemonize(argv[0], 0);
//syslog(LOG_NOTICE|LOG_USER, "ISALIVE!!!!!!!!!!! bruno");

if( (retCode = initializeSingleTcpServer(PORTNUMBER, &socketDescriptor, 1)) < 0)
{
writeLog(tcpipErrorMsg[retCode]);
return(retCode);
}
#if DEBUG_STA
int count = 0; //just a DEBUG_STA VAR
#endif


while(1)
{
#if DEBUG_STA
cout << ":::::::::::::::::::::::::::::::::::::: (" << count++ << ")" << endl;
#endif
fflush(stdout);
sin_size = sizeof(client_addr);
if ((new_fd = accept(socketDescriptor, (struct sockaddr *) &client_addr, &sin_size)) < 0)
{
printf("Erro no new_fd\n");
continue;
}
#if DEBUG_STA
printf("TCP/IP server      : Connection accepted from %s\n", inet_ntoa(client_addr.sin_addr));
#endif

/* memset(buf, '\0', DATA_SIZE);
strncat(buf, "TCP/IP server      : Connection accepted from ", strlen("TCP/IP server      : Connectic
```

```
strncat(buf, inet_ntoa(client_addr.sin_addr), strlen(inet_ntoa(client_addr.sin_addr)));
writeLog(buf);
*/

retCode = answerProbe(key, new_fd);
if(retCode)
{
memset(buf, '\0', DATA_SIZE);
strncat(buf, "Prober closed with error:", strlen("Prober closed with error:"));
strncat(buf, (const char*)retCode, 1);
writeLog(buf);

close(new_fd);
exit(-1);
}
#if DEBUG_STA
cout << "::::::::::::::::::::::::::::::::: close(new_fd)" << endl;
#endif
close(new_fd);

}// END OF WHILE(1)...

writeLog("*** THE IS_ALIVE_STATION IS NOW DEAD!!! ***");
writeLog("*** HAVE A NICE DAY!!! ***");

#if DEBUG_STA
cout << "\n\n *** THE IS_ALIVE_STATION IS NOW DEAD!!! *** \n" << endl;
cout << "\n\n *** HAVE A NICE DAY!!! *** \n" << endl;
#endif

return (retCode);

}// END OF main()
//=====================================================================================



//=====================================================================================
// This function writes log_msg in the log file.
```

```
void writeLog(char *log_msg)
{
FILE *out;
struct tm *ptr;
time_t lt;
char buf[DATA_SIZE];
memset(buf, '\0', DATA_SIZE);

lt = time(NULL);
ptr = localtime(&lt);
strncat(buf, asctime(ptr), 24);
strncat(buf, ": ", 2);
strncat(buf, log_msg, strlen(log_msg));

out = fopen(LOG_FILE, "a");
fwrite(buf, 1, 26+strlen(log_msg), out);
fwrite("\n", 1, 1, out);
fclose(out);

}// END OF writeLog()
//==============================================================================



//==============================================================================
// Initializes the current program as a daemon, by changing working
// directory, umask, and eliminating control terminal,
// setting signal handlers.
void  daemonize(const char *pname, int facility)
{
  pid_t pid;
  int k;

  /* put server in background (with init as parent) */
  if ( ( pid = fork() ) < 0 ) {
    perror(">>> cannot fork2");
    exit(1);
  } else if (pid > 0) /* The parent */
    exit(0);
```

```
  /* Detach controlling terminal by becoming sesion leader */
  setsid();

  signal(SIGHUP, SIG_IGN);
  /* put server in background (with init as parent) */
  if ( ( pid = fork() ) < 0 ) {
    perror(">>> cannot fork2");
    exit(1);
  } else if (pid > 0) /* The parent */
    exit(0);

  fclose(stdin);
  fclose(stderr);
  fclose(stdout);

  /* Close all file descriptors that are open */
  for (k = MAX_FD; k>0; k--)
      close(k);

  /* Change directory to specified directory */
//  chdir("/");

  /* Set umask to mask (usually 0) */
  umask(0);

  openlog(pname, LOG_PID, facility);

}// END OF daemonize()
//======================================================================================


//======================================================================================
// This function probes the station
ulong answerProbe(unsigned char *firstKey, int new_fd)
{
/*  CRYPT  ---------------------------------------------------*/
int outlen = 0;
```

```
int tmplen = 0;
unsigned char iv[] = {1,2,3,4,5,6,7,8};
EVP_CIPHER_CTX ctx;
/*  CRYPT  --------------------------------------------------*/
ulong retCode = OK;
int bytes_recved = 0;
int bytes_sent = 0; // how many bytes we´ve sent at that moment

byte *txmsg = 0;
txmsg = new byte[DATA_SIZE];
memset(txmsg, '\0', DATA_SIZE);

byte *rxmsgDecrypted = 0;
rxmsgDecrypted = new byte[DATA_SIZE];
memset(rxmsgDecrypted, '\0', DATA_SIZE);

byte *txmsgCrypted = 0;
txmsgCrypted = new byte[DATA_SIZE];
memset(txmsgCrypted, '\0', DATA_SIZE);

byte *rxmsgCrypted = 0;
rxmsgCrypted = new byte[DATA_SIZE];
memset(rxmsgCrypted, '\0', DATA_SIZE);

#if DEBUG_STA
cout << "FirstKey ->:" << firstKey << ":<-" << endl;
cout << "NEW_FD   ->:" << new_fd   << ":<-" << endl;
cout << "==============================================================" << endl;
#endif

  // Recive first msg from the isAliveDaemon
if( ( bytes_recved = recv(new_fd, rxmsgCrypted, DATA_SIZE, 0) ) < 0 )
return(ERROR_REVC);

// Decrypt message recived from isAliveDaemon -------------------------------------------------
#if DEBUG_STA
cout << "==>bytes_recved:" << bytes_recved << endl;
#endif
```

```
EVP_CIPHER_CTX_init(&ctx);
EVP_DecryptInit_ex(&ctx, (EVP_CIPHER*)EVP_bf_cbc(), NULL, firstKey, iv);
if( !EVP_DecryptUpdate(&ctx, rxmsgDecrypted, &outlen, rxmsgCrypted, 24) )
{
/* Error */
cout << "Error in EVP_DecryptUpdate()!" << endl;
return 1;
}
/* Buffer passed to EVP_EncryptFinal() must be after data just
 * encrypted to avoid overwriting it.*/
EVP_CIPHER_CTX_set_padding(&ctx, 0);
if(!EVP_DecryptFinal_ex(&ctx, rxmsgDecrypted + outlen, &tmplen))
{
/* Error */
cout << "Error in EVP_DecryptFinal_ex()!" << endl;
return 1;
}
outlen += tmplen;
EVP_CIPHER_CTX_cleanup(&ctx);
/* Need binary mode for fopen because encrypted data is
 * binary data. Also cannot use strlen() on it because
 * it wont be null terminated and may contain embedded
 * nulls.
 */
// END OF Decrypt message recived from isAliveDaemon --------------------------
#if DEBUG_STA
fflush(stdout);
cout << "strlen(rxmsgDecrypted): " << strlen((char*)rxmsgDecrypted) << endl;
fflush(stdout);
cout << "outlen                : " <<  outlen  << endl;
fflush(stdout);
cout << "\nrxmsgCrypted          :"  << rxmsgCrypted  << ":" << endl;
fflush(stdout);
cout << "rxmsgDecrypted        :"  << rxmsgDecrypted << ":" << endl;
fflush(stdout);
cout << "Decrypted message     :" << rxmsgDecrypted << endl;
fflush(stdout);
```

```
cout << "OLD KEY               :" << key << endl;
fflush(stdout);
#endif

memmove(key, rxmsgDecrypted, KEY_SIZE);

#if DEBUG_STA
cout << "rxmsgDecrypted        :" << rxmsgDecrypted << endl;
fflush(stdout);
cout << "NEW KEY               :" << key << endl;
#endif

memset(txmsg, '\0', DATA_SIZE);
memset(txmsgCrypted, '\0', DATA_SIZE);
strcpy((char*)txmsg, ACK_MSG);
strcat((char*)txmsg, "\0");  //0x00;  //EOS

// Encrypt message to be sent to isAliveDaemon --------------------------------------------------
EVP_CIPHER_CTX_init(&ctx);
EVP_CIPHER_CTX_set_padding(&ctx, 1);
EVP_EncryptInit_ex(&ctx, (EVP_CIPHER*)EVP_bf_cbc(), NULL, key, iv);
if(!EVP_EncryptUpdate(&ctx, txmsgCrypted, &outlen, (unsigned char*)txmsg, strlen((char*)txmsg)))
{
/* Error */
return 1;
}
/* Buffer passed to EVP_EncryptFinal() must be after data just
 * encrypted to avoid overwriting it.
 */
if(!EVP_EncryptFinal_ex(&ctx, txmsgCrypted + outlen, &tmplen))
{
/* Error */
return 1;
}
outlen += tmplen;
EVP_CIPHER_CTX_cleanup(&ctx);
/* Need binary mode for fopen because encrypted data is
 * binary data. Also cannot use strlen() on it because
```

```
 * it wont be null terminated and may contain embedded
 * nulls.
 */
// END OF Encrypt message to be sent to isAliveDaemon --------------------
#if DEBUG_STA
cout << "==>txmsg=ACK_MSG:" << txmsg << ":" << endl;
cout << "==>txmsgCrypted:" << txmsgCrypted << ":" << endl;
#endif

// sending encrypted ACK message to the client
bytes_sent = send(new_fd, txmsgCrypted, outlen, 0);
if (bytes_sent < 0)
{
cout << tcpipErrorMsg [retCode] << endl;
return(ERROR_SEND);
}

#if DEBUG_STA
// printing message sent to isAliveDaemon
cout << "Message sent to prober:" << txmsgCrypted << endl; //myServer.getTXmsg(0) << endl;
cout << "================================================================" << endl;
#endif

fflush(stdout);
delete [] txmsg;
delete [] rxmsgDecrypted;
delete [] txmsgCrypted;
delete [] rxmsgCrypted;

#if DEBUG_STA
cout << "End of answerProbe()! Returning OK! " << endl;
#endif
return(OK);

}// END OF answerProbe()
//=========================================================================
```

```
//==========================================================================================
// This function initializes a tcp server calling socket, bind, listen
ulong initializeSingleTcpServer(ushort port, int *socketDescriptor, int verbose)
{
        struct  sockaddr_in server_addr; //Hold the destination addr
        struct  hostent *h; //Hold the server information
        char    serverAddress[SIZE_DATA];
        int     sock_fd;
        int     yes = 1;
char buf[DATA_SIZE]; memset(buf, '\0', DATA_SIZE);
char convert[10];

        #if DEBUG_STA
                printf("DEBUG_STA: IN initializeSingleTcpServer FUNCTION \n");
        #endif

        if((gethostname(serverAddress, sizeof(serverAddress))) < 0) // get the host name
                return(ERROR_GETHOSTNAME);

        if((h = gethostbyname(serverAddress)) == NULL) // get the host info
                return(ERROR_GETHOSTBYNAME);

        // Setting structure parameters ************************
        server_addr.sin_family = SOCK_FAMILY;
        server_addr.sin_port   = htons(port);
        //server_addr.sin_addr   = *((struct in_addr *)h->h_addr);
        server_addr.sin_addr.s_addr  = htonl(INADDR_ANY);
        memset(&(server_addr.sin_zero), '\0', 8); // zero to the rest of the structure
        // Setting structure parameters *******END**************

        // Opening socket ************************
        if ((sock_fd = socket(SOCK_FAMILY, SOCK_TYPE, SOCK_PROTOCOL)) < 0)
                return(ERROR_OPEN_SOCKET);

if(verbose)
printf("TCP/IP server      : Socket Opened ... \n");
```

```
        // Forcing the port to be released ********
        // Lose the pesky "Address alredy in use" error message *********
        setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));

        // Binding to port *******************
        if ((bind(sock_fd, (struct sockaddr *) &server_addr, sizeof(server_addr))) < 0)
        {
                close(sock_fd);
                return(ERROR_BIND);
        }

if(verbose)
         printf("TCP/IP server      : Bind to port %d ...\n", port);

        // Listenig to port *******************
        if ((listen(sock_fd, BACKLOGSINGLE)) < 0)
        {
                close(sock_fd);
                return(ERROR_LISTEN);
        }



if(verbose)
{
         printf("TCP/IP server      : Listening port %d ...\n", port);
         printf("Server Name (IP)   : %s (%s)\n\n", h->h_name, inet_ntoa(*((struct in_addr *)h->h_add
}

memset(buf, '\0', DATA_SIZE);
strncat(buf, "TCP/IP server      : Listening port ", strlen("TCP/IP server      : Listening port "));
sprintf(convert, "%d", port);
strncat(buf, convert, strlen(convert));
writeLog(buf);

memset(buf, '\0', DATA_SIZE);
strncat(buf, "Server Name (IP)   : ", strlen("Server Name (IP)   : "));
strncat(buf, h->h_name, strlen(h->h_name));
strncat(buf, "(", 1);
```

```
strncat(buf, inet_ntoa(*((struct in_addr *)h->h_addr_list[0])), strlen(inet_ntoa(*((struct in_addr *)
strncat(buf, ")", 1);
writeLog(buf);

*socketDescriptor = sock_fd;

        return(OK);
}// END OF initializeSingleTcpServer
//==================================================================================
```

Apêndice C

Código Fonte do isAliveDaemon

```
// isAliveDeamon.h (v1.0) *** Apenas para SOs UNIXlike ***
// Desenvolvido sobre plataforma OpenBSD 3.3
// Author: Bruno Astuto Arouche Nunes
// Date: 17-01-2004
// 2004/1
//
// Compilando
// g++ -I/usr/local/include/ -I/usr/local/include/mysql -O2 -c isAliveDaemon.cpp tcp_lib.cpp
//
// Linkando
// g++ -g -O2 -L/usr/local/lib/mysql -o isAliveDaemon isAliveDaemon.o -L/usr/local/lib/
// -lsqlplus -lz -lmysqlclient -lcrypto tcp_lib.cpp
//-----------------------------------------------------------------------------------------

#ifndef _ISALIVEDAEMON_H_
#define _ISALIVEDAEMON_H_

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <fstream.h>
#include <iostream.h>
#include <sys/wait.h>
#include <signal.h>
#include <syslog.h>
#include <iomanip.h>
```

```
#include <openssl/evp.h>
#include "sqlplus.hh"
#include "tcp_lib.h"

#define VERBOSE 0
#define PROBE_ERROR 13
#define UPDATE_ERROR 17
#define TIME_TO_WAIT 1
#define PORTNUMBER 56789
#define KEY_SIZE 16 // 16 char = 128 bits
#define LINE_SIZE 4096
#define MAX_TTL 2 // max number of times the daemon wil try to probe the station
#define ACK_MSG_SIZE 7
#define ACK_MSG "200 ACK"
#define TOKEN ":"
#define CONF_FILE_NAME "isAliveDaemon.conf"
#define DB_NAME "DB_NAME"
#define DB_HOST "DB_HOST"
#define DB_USER "DB_USER"
#define DB_PASSWD "DB_PASSWD"
#define DB_TABLE_NAME "DB_TABLE_NAME"


/* -------------- PROTOTYPES --------------- */
// This function is responsable for the connection with the STAs. Its called by queryandprobe(con);
ulong probeStation(char *serverAddr, int port, unsigned char *key, char *tableName);

// Reload Packet Filter's rules *** OpenBSD 3.3 ONLY!!! ***
ulong reloadPf(char *staAddr, char *tableName);

// Read parameters from the conf file.
void readConfFile();

// Key generator : this is not the best way to do this.
// it would be nice to find another way of generating keys.
//-----------------------------------------------------------
void seedGen();
char* getKey();
```

```
//--------------------------------------------------------

// This function connects to the batabase and gets the info needed
// to probe the STAs with the probeStation(...) function.
int queryandprobe(Connection *con);
/* ----------- END OF PROTOTYPES ------------ */

#endif
```

Apêndice D

Script para Criação do BD StrikeIN

```
Script para criação do BD StrikeIN
CREATE TABLE ipsValidos ( senhaAtual VARCHAR(20) NOT NULL, login VARCHAR(20)
NOT NULL, enderecoIP VARCHAR(16) NOT NULL, TTL int(2) NOT NULL );
ALTER TABLE ipsValidos ADD PRIMARY KEY (login, enderecoIP);
CREATE TABLE passwd ( senha VARCHAR(20) NOT NULL, login VARCHAR(20) NOT NULL,
nome VARCHAR(20) NOT NULL ); ALTER TABLE passwd ADD PRIMARY KEY (login);
ALTER TABLE ipsValidos ADD FOREIGN KEY (login) REFERENCES passwd (login) ON
DELETE RESTRICT ON UPDATE RESTRICT;
```

Referências Bibliográficas

[1] Demetrio S. D. Carrión, "Implementação de um ponto de acesso seguro para redes 802.11b baseado no sistema operacional OpenBSD," *Projeto Final de Curso, Departamento de Engenharia Eletrônica e Computação - DEL/UFRJ*, 2003.

[2] P. D. M. Júnior, B. A. A. Nunes, C. A. V. Campos, and L. F. M. de Moraes, "Avaliando a Sobrecarga Introduzida nas Redes 802.11 pelos Mecanismos de Segurança WEP e VPN/IPSec," *WSEG2003/SBRC2003, Natal/RN*, maio 2003.

[3] 802.11b, *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification: Higher-Speed Physical Layer Extension in the 2.4 GHz Band*. IEEE Std 802.11b, 1999.

[4] J. Walker, "Unsafe at any key size: an analysis of the WEP encapsulation," tech. rep., IEEE 802.11 committee, March 2000.

[5] N. Borisov, I. Goldberg, and D. Wagner, "Intercepting Mobile Communications: The Insecurity of 802.11," *http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html*.

[6] W. A. Arbaugh, N. Shankar, and Y. C. J. Wan, "Your 802.11 Wireless Network has No Clothes," Tech. Rep. 03628E, University of Maryland, March 30 2001.

[7] OpenBSD, "Sistema operacional de código aberto e gratuito baseado no BSD 4.4." http://www.openbsd.org, Página visitada em 2003.

[8] OpenSSL, "Projeto OpenSSL que implementa o SSL e o TSL." http://www.openssl.org, Página visitada em 2003.

[9] G. Huang, S. Beaulieu, D. Rochefort, and IPSec Working Group, "A Traffic-Based Method of Detecting Dead IKE Peers," *IETF Internet Draft: draft-ietf-ipsec-dpd-00.txt*, July 2001.

[10] S. Fluhrer, I. Mantin, and A. Shamir, "Weaknesses in the key scheduling algorithm of RC4," *Eighth Annual Workshop on Selected Areas in Cryptography*, 2001.

[11] *RC4: Encrypty Algoritm of RSA Security*, 2003. http://www.rsasecurity.com.

[12] *Lucent Orinoco - User's Guide for the ORiNOCO Manager's Suite*, 2000. Lucent Orinoco.

[13] P. Srisuresh and K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)," *IETF RFC 3002*, 2001.

[14] S. Kent and R. Atkinson, "IP Authentication Header," *IETF RFC 2402*, nov 1998.

[15] S. Kent and R. Atkinson, "IP Encapsulating Security Payload (ESP)," *IETF RFC 2406*, nov 1998.

[16] "Oasis - ferramenta de autenticação centralizada.," 2002. http://software.stockholmopen.net.

[17] "Nocat - ferramenta de autenticação centralizada.," 2002. http://www.nocat.org.

[18] "Netlogon - ferramenta de autenticação centralizada.," 2002. http://unit.liu.se/dokument/natverk/netlogon.html.

[19] R. Stevens, *Network Programming for Unix*. PH PTR, 2nd ed., 1996.

[20] Cygwin, "Linux-like environment for windows." http://www.cygwin.com, Página visitada em 2003.

[21] D. Knuth, *The Art of Computer Programming*. Addison-Wesley, 2nd ed., 1981.

[22] S. J. S. D. Eastlake 3rd, "Randomness Recommendations for Security." RFC1750, dec 1994.

[23] G. V. L. F. M. d. M. Alexandre Mendes, Bruno Astuto Arouche Nunes, "Estudo da Implementação e Avaliação de Algoritmos de Geração de Seqüências Aleatórias e suas Aplicações," 2003.

[24] MYSQL++, "Api c++ para acesso ao banco de dados mysql." http://www.mysql.com/products/mysql++/, Página visitada em 2004.