ADEMAR DE SOUZA REIS JUNIOR
MILTON SOARES FILHO

UM SISTEMA DE TESTES PARA A DETECÇÃO REMOTA DE *SNIFFERS* EM REDES TCP/IP

Trabalho de Graduação apresentado ao Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto André Hexsel

CURITIBA

2002

Agradecimentos

Nossos sinceros agradecimentos a nosso orientador, professor Dr. Roberto André Hexsel, que foi nosso paciente guia durante toda a confeccão deste texto. Agradecimentos também aos professores Dr. Elias Procópio Duarte Jr. e Armando Luiz Nicolini Delgado e a Andreas Hasenack por suas revisões, dicas e sugestões. Estendemos nossos agradecimentos ainda a toda a comunidade de software livre, que nos propiciou excelentes ferramentas e bibliotecas para o desenvolvimento e testes de nosso projeto.

Sumário

Lista de Figuras **iv**

RESUMO **v**

ABSTRACT **vi**

1 Introdução **1**

2 *Sniffers* **3**

2.1 Redes de Difusão (Meio Compartilhado) 4

2.2 Redes Comutadas 4

2.3 Modos de Operação da Interface de Rede 4

2.4 Utilização em Ataques 5

2.4.1 Ataque Inicial 5

2.4.2 Anatomia de Ataques com o Uso de *Sniffers* 6

2.5 Evitando o uso Efetivo de *Sniffers* 9

2.5.1 Limitação da Visibilidade do Tráfego 9

2.5.2 Utilização de Interfaces que Não Suportem Modo Promíscuo 9

2.5.3 Utilização de Criptografia 10

2.5.4 Detecção 12

3 Detecção de *sniffers* **13**

3.1 Detectores de Intrusão 13

3.2 Detecção Local 14

3.3 Detecção Remota 14

3.3.1 Requisição ICMP com Falso Endereço Físico 15

3.3.2 Requisição ARP com Falso Endereço Físico 16

3.3.3 DNS Reverso 17

3.3.4 Latência 17

3.3.5 Armadilha *(Honey Pot)* 19

3.3.6 Detecção de Inundação de Respostas ARP 20

3.4 Confiabilidade dos Testes 20

4 Implementação **21**
4.1 Preocupações do Projeto . . . 21
4.2 Documentação . . . 22
4.3 Licença e Distribuição . . . 22
4.4 Ambientação . . . 23
4.5 Biblioteca . . . 24
4.5.1 Arquitetura Geral . . . 25
4.5.2 Funcionamento . . . 27
4.5.3 Responsividade . . . 27
4.5.4 Resultado dos Testes de Detecção . . . 28
4.5.5 Teste ICMP . . . 29
4.5.6 Teste ARP . . . 30
4.5.7 Teste DNS . . . 30
4.5.8 Teste de Latência . . . 31
4.6 Aplicação . . . 33
4.6.1 Interface texto . . . 34
4.6.2 Plugins de Relatórios de Resultados . . . 35
4.6.3 Arquivo de Configuração . . . 36
4.6.4 Segurança . . . 36
4.7 Considerações Pós-desenvolvimento . . . 36

5 Experimentos e Resultados **38**
5.1 Ambientação . . . 38
5.2 Descrição dos Resultados . . . 39
5.2.1 Teste ICMP . . . 39
5.2.2 Teste ARP . . . 40
5.2.3 Teste DNS . . . 41
5.2.4 Teste de Latência . . . 42
5.2.5 Comportamentos Inesperados . . . 44

6 Conclusão **46**

A Páginas de Manual *("Unix Manpages")* **48**
A.1 libsniffdet (3) . . . 48
A.2 sniffdet (1) . . . 54
A.3 sniffdet.conf (2) . . . 56

Referências Bibliográficas **58**

Lista de Figuras

2.1 Arquitetura de um sniffer. 3
2.2 Sniffer em ação em uma rede de difusão. 6
2.3 Sniffer ligado a um servidor. 7
2.4 Atacante utilizando-se de técnicas de redirecionamento para capturar tráfego em uma rede comutada. 8

4.1 Arquitetura geral da biblioteca. 26
4.2 Funcionamento das threads dos testes de detecção. 27
4.3 Arquitetura Geral da Aplicação 33

Resumo

Sniffers são ferramentas utilizadas para capturar e opcionalmente analisar tráfego de rede. Este trabalho faz uma análise do funcionamento dessas ferramentas, o potencial perigo de seu uso do ponto de vista da segurança de uma rede e técnicas de defesa e detecção. Também descreve a implementação aberta e de livre distribuição de uma biblioteca e de uma aplicação para a detecção remota de *sniffers* em redes Ethernet, assim como os resultados obtidos com seus testes.

Abstract

Sniffers are tools used to capture and optionally analyze network traffic. This work discusses their behaviour, potential security risks when used by malicious users in a network and available defense and detection techniques. It also describes the implementation of an open source library and an application for remote sniffer detection in Ethernet networks, and the results of the experiments performed.

Capítulo 1

Introdução

O assunto “segurança computacional” tem sido muito discutido nos últimos tempos. Como parte importante deste tópico temos as medidas de contenção, cujo objetivo é evitar que invasões ocorram, e as medidas de detecção, que procuram por indícios de invasões considerando uma possível falha nas barreiras levantadas pelas medidas de contenção. Nesta última categoria entram os conhecidos detectores de intrusão e a perspicácia dos administradores de sistemas.

Uma das ferramentas mais utilizadas por atacantes é o *sniffer*, um software usado para capturar e analisar tráfego de rede. Em ataques simples ou sofisticados, o uso de um *sniffer* é parte importante de sua concepção e efetivação. Com a ainda ampla utilização de protocolos não criptografados, a utilização de um *sniffer* pode revelar dados consideravelmente sensitivos e importantes para os atacantes.

É importante conhecer a arquitetura e topologia das redes onde um *sniffer* pode atuar. Porém, independentemente das características desta, a utilização de um *sniffer* por um atacante não deixa de ser um grande risco à sua segurança. Neste trabalho são estudadas diversas arquiteturas e topologias, assim como os riscos e potenciais vulnerabilidades destas, em especial quando se tem um *sniffer* como ferramenta de ataque a ser utilizada.

Devido a sua operação inerentemente passiva e o potencial comprometimento prévio da máquina utilizada, tanto a detecção local como a remota de um *sniffer* tornam-se tarefas difíceis. Através da compreensão de seu funcionamento e do estudo aprofundado das características das diferentes implementações da pilha TCP/IP é possível planejar técnicas que apontem sua utilização não autorizada no ambiente de rede testado, mesmo que de forma não determinística. Este trabalho contém uma análise das diferentes técnicas de detecção existentes, mostrando seus pontos fracos e fortes.

A falta de uma ferramenta flexível, de código fonte aberto e de livre distribuição motivou a implementação dos testes discutidos. Os resultados do trabalho aqui exposto são uma biblioteca de testes e uma ferramenta que visa suprir as expectativas de administradores de redes, profis-

sionais de segurança e estudantes da área de redes. Tal implementação é voltada para redes que seguem o padrão Ethernet, o mais utilizado entre os padrões para redes locais.

No capítulo 2, discorremos sobre os *sniffers*, suas origens, utilização e implicações práticas na segurança das redes em que são executados. No capítulo 3, são mostradas e comentadas diversas técnicas para a detecção de *sniffers*. O capítulo 4 descreve a implementação de uma biblioteca de testes de detecção e uma aplicação que a utiliza. Por fim, temos os resultados dos testes realizados com a utilização de tal sistema no capítulo 5 e as conclusões do trabalho no capítulo 6.

Capítulo 2

Sniffers

Sniffers[1] são ferramentas utilizadas para capturar e opcionalmente analizar tráfego de rede. Estes são tão antigos quanto as redes de computadores e sua função originalmente foi a de ajudar desenvolvedores de protocolos e administradores de rede a solucionarem problemas, diagnosticar falhas e levantar dados estatísticos.

Um *sniffer* pode ser utilizado de diversas maneiras e com diversos propósitos, mas seu princípio de funcionamento continua sendo o mesmo: capturar e análizar tráfego de rede sem interferir no funcionamento desta. A arquitetura de um *sniffer* pode ser vista na figura 2.1.

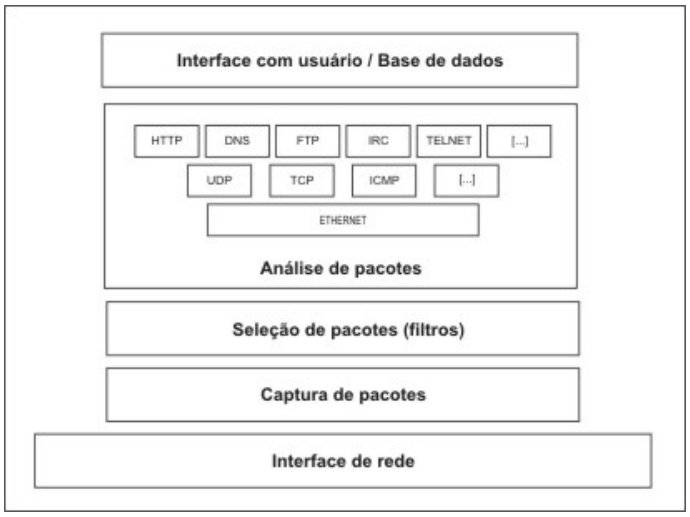

Figura 2.1: *Arquitetura de um sniffer.*

As mais variadas ferramentas se encaixam no conceito de *sniffers*. Entre elas, existem as benéficas, como analisadores de tráfego, utilizados para estudo e depuração de protocolos (como por exemplo o Ethereal [16] e o tcpdump [38]), detectores de intrusão, que procuram por

[1] O termo "sniffer", que em inglês significa "farejador" é uma marca registrada da empresa Network Associates, referindo-se ao produto "Sniffer Network Analyser". Atualmente o termo é utilizado ampla e genericamente em referência a qualquer dispositivo ou software cuja função seja capturar tráfego de redes.

assinaturas de ataques no tráfego da rede (como o Snort [37]) e as que geralmente são utilizados por atacantes ou usuários mal intencionados com o objetivo de capturar senhas e informações relevantes a partir do tráfego da rede.

A topologia da rede, a interface de comunicação e sua operação, os protocolos utilizados e o comportamento dos usuários são fatores importantes ao analisar-se o potencial de uso de um *sniffer*, principalmente quando o objetivo é a captura de informações alheias. Esses fatores são discutidos no decorrer deste capítulo.

2.1 Redes de Difusão (Meio Compartilhado)

Redes de difusão são caracterizadas pelo compartilhamento do meio de transmissão de dados, que é a camada enlace da rede. Seu uso é comum em configurações de pequeno porte, como redes domésticas e de pequenos laboratórios e escritórios, já que seu custo de implementação é baixo. Um caso particular de rede de difusão é o das redes sem fio, comumente chamadas de "wireless". Nestas, o compartilhamento do meio – espaço físico – é aberto e de difícil limitação.

Topologias como "estrela" e "barramento" são exemplos típicos de redes de difusão. Em redes que seguem o padrão Ethernet, extremamente popular e presente em grande parte das instalações de redes locais, são utilizados *hubs* ou cabos coaxiais para a implementação de redes de difusão.

2.2 Redes Comutadas

Redes comutadas por sua vez fazem uso de um enlace dedicado para cada máquina ligada à rede. O tráfego é distribuído com base no endereço de destino dos pacotes através de um comutador (um "switch" no caso de redes Ethernet). Seu desempenho é superior quando comparado às redes de difusão mas seu custo de implementação é mais alto, dada a necessidade de hardware especializado.

2.3 Modos de Operação da Interface de Rede

No modo de operação normal, uma interface de rede deve descartar o tráfego de rede que não é a ela direcionado. É possível alterar o modo como uma interface de rede trabalha e fazer com que todo o tráfego que passe pelo meio de transmissão seja capturado, não importando a quem ele é destinado. Tal modo é chamado de "modo promíscuo". Nas interfaces de comunicação

que seguem o padrão Ethernet, sua implementação faz parte da especificação e geralmente pode ser habilitado através de comandos de software.

Uma máquina deve capturar apenas o tráfego a ela endereçado, descartando os pacotes alheios. Essa seleção de pacotes geralmente é feita em nível de hardware ou *firmware*, evitando assim que as camadas superiores da pilha de rede, implementadas em software, tenham que se preocupar com tal seleção.

Um *sniffer* pode ser utilizado com a interface de rede em qualquer modo de operação. Sua eficiência em recolher dados porém é maior quando a interface está em modo promíscuo. Uma máquina com a interface de rede em modo promíscuo conectada a uma rede de difusão consiste no cenário ótimo para um *sniffer*, na qual todo o tráfego na rede pode ser capturado e analisado.

2.4 Utilização em Ataques

Como ferramenta poderosa que é, não demorou para que usuários mal intencionados começassem a utilizar *sniffers* para a coleta de dados privilegiados, transformando-os numa das principais ferramentas utilizadas em ataques a redes de computadores. Um *sniffer* tem o potencial de revelar dados críticos de uma organização tais como senhas, números de cartões de crédito, correspondências, documentos, enfim, toda e qualquer informação que trafegue de forma desprotegida pela rede.

2.4.1 Ataque Inicial

Normalmente, o processo de invasão de uma rede alheia começa quando o atacante obtém algum tipo de privilégio de administrador em uma máquina dessa rede. Isso pode ser conseguido de diversas maneiras, como com a exploração de vulnerabilidades remotas em software da rede, uso de vírus, acesso físico irrestrito à máquinas da rede ou ao meio transmissor de dados e, muito comumente, através da utilização de técnicas de "engenharia social" [32, 43], nas quais o atacante obtém acesso ou informações relevantes sobre o sistema através de contato pessoal, muitas vezes fazendo-se passar por outra pessoa.

Têm sido relatados os mais inesperados casos de ataques de engenharia social, como por exemplo atacantes que se passam por funcionários da empresa (mesmo que um inocente faxineiro), analistas de suporte, fiscais de segurança, professores, etc. Truques telefônicos e conversas através de meios eletrônicos em geral são fontes comuns de ataques de engenharia social [7].

Estatísticas mostram que agentes já familiarizados com o ambiente da rede, como os próprios funcionários de uma empresa ou pessoas com acesso físico irrestrito são as princi-

pais portas de entrada no ataque a uma rede. Em pesquisa feita entre empresas brasileiras no primeiro semestre de 2001, 53% dos entrevistados apontaram funcionários insatisfeitos como a maior ameaça à segurança da informação, enquanto que, dos ataques registrados, cerca de 34% se originam de funcionários, fornecedores ou prestadores de serviço [33].

Uma vez que um atacante tenha privilégios de administrador em uma máquina conectada à rede, a instalação de um *sniffer* e outras ferramentas de ataque (como os chamados *rootkits*[2]) é muito simples e o comprometimento das informações passa a ser uma mera questão de tempo.

2.4.2 Anatomia de Ataques com o Uso de *Sniffers*

Existem diversos cenários e topologias nas quais *sniffers* podem ser utilizados para a captura de informações alheias. Técnicas de proteção e evasão desses ataques são discutidas na próxima seção.

Ataques a Redes de Difusão

Redes de difusão apresentam o cenário ótimo para um *sniffer* cujo objetivo seja a captura de tráfego alheio. Uma vez que este tráfego é difundido por todo o barramento, um *sniffer* localizado em qualquer ponto da rede é capaz de capturá-lo por completo simplesmente utilizando-se de uma interface de rede em modo promíscuo. Tal cenário é exemplificado na figura 2.2.

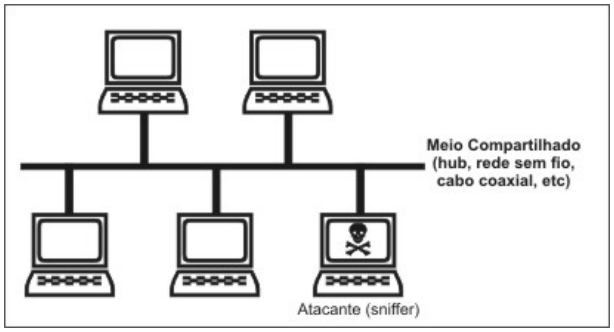

Figura 2.2: *Sniffer em ação em uma rede de difusão.*

Ataques a Redes Comutadas

Uma vez que em redes comutadas o meio não é compartilhado entre todas as máquinas, a localização do *sniffer* torna-se um fator muito importante. A instalação de um *sniffer* em um

[2]Um *rootkit* consiste-se de um conjunto de ferramentas que auxiliam o atacante a manter-se em sigilo e dominar a máquina por completo. Podem ser facilmente encontrados na Internet e geralmente incluem módulos de kernel, *backdoors*, *sniffers* e demais ferramentas úteis a um atacante.

servidor ou roteador consite no cenário ideal, pois este pode capturar o tráfego ali canalizado, como exemplificado na figura 2.3, onde o *sniffer* está no mesmo enlace que uma máquina importante.

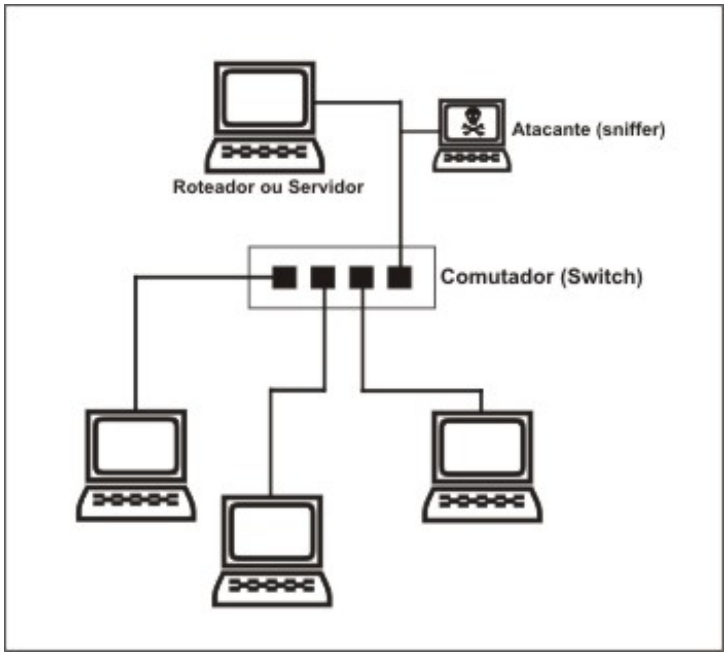

Figura 2.3: *Sniffer ligado a um servidor.*

Nessas redes, através da utilização de técnicas que visam "enganar" diversos protocolos, é possível a um atacante capturar até **todo** o tráfego da rede mesmo tendo acesso apenas a uma máquina. Neste cenário, o atacante tenta enganar as máquinas da rede de forma que o tráfego seja redirecionado para um local onde possa ser capturado. Agindo como um *proxy*, o atacante mantém o fluxo normal da rede, evitando que os usuários percebam sua presença. Uma análise e exemplos deste tipo de ataque podem ser vistos em [40].

A figura 2.4 exemplifica o caso em que uma máquina é enganada, tendo seu tráfego roteado através da máquina de um atacante.

Casos Particulares

- Dispositivos não convencionais

  A utilização de dispositivos não convencionais vem se tornando popular entre os atacantes. Pela aparência inofensiva que têm, consoles de videogames (que ultimamente saem de fábrica equipados com suporte à rede) vêm sendo utilizados em ataques a redes corporativas[3]. Além disso, nada impede que um atacante crie seu próprio dispositivo eletrônico

[3]Hackers já usam consoles em invasões - `http://www2.uol.com.br/info/aberto/infonews/082002/02082002-10.shl`. Attack Of The Dreamcasts - `http://slashdot.org/article.pl?sid=02/08/01/1535234&mode=nested&tid=172`.

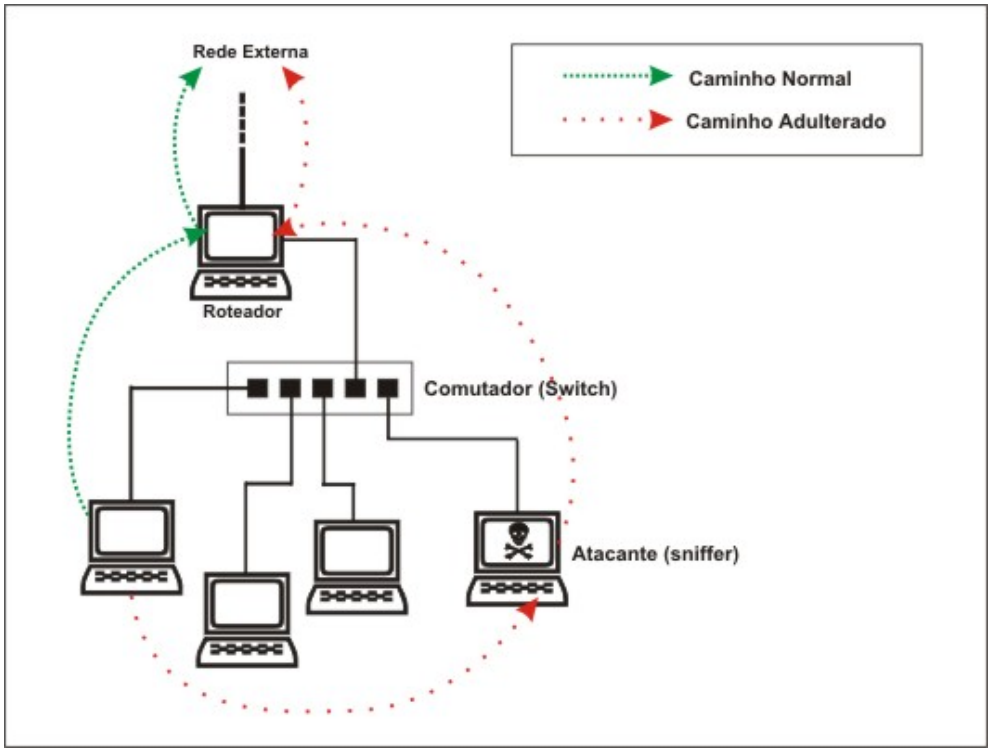

Figura 2.4: *Atacante utilizando-se de técnicas de redirecionamento para capturar tráfego em uma rede comutada.*

capaz de capturar tráfego da rede, precisando então apenas ter acesso direto ao meio físico para sua instalação.

- Redes sem fio

  Em redes sem fio, utilizar um *sniffer* é muito simples, pois basta ao atacante conseguir uma certa proximidade de seu alvo e utilizar-se de um equipamento que opere na mesma frequência da rede em questão. Tem sido comum encontrar atacantes utilizando-se de potentes antenas caseiras (que podem feitas com latas de batata frita) para capturar tráfego de redes sem fio em grandes centros comerciais[4].

- Análise ótica

  Como se não bastasse a captura do tráfego através do meio de transmissão, um recente estudo mostra que é possível a captura do tráfego de uma rede através da análise de luzes de *switchs* e *hubs* a distâncias de até 30 metros [22].

[4]Também conhecidos como *wardrivings*, que consistem na procura por redes sem fio através do uso de algum meio de locomoção.

2.5 Evitando o uso Efetivo de *Sniffers*

Existem várias maneiras de se evitar o uso efetivo de um *sniffer* por parte de um atacante. Embora não exista uma "fórmula mágica" ou método totalmente eficaz, existem várias técnicas que podem ser utilizadas na luta contra os atacantes.

O uso de canais de comunicação criptografados, embora não evite o uso de *sniffers*, é a técnica mais eficaz para a proteção de informações na rede, pois torna o tráfego incompreensível a quem não conheça a chave para descriptografia. O uso de hardware especializado, técnicas de detecção (remota e local) e intenso trabalho de administração mostram-se efetivos para vários casos, mas nem sempre são confiáveis.

Abaixo fazemos uma análise de várias dessas técnicas, apontando seus prós e contras.

2.5.1 Limitação da Visibilidade do Tráfego

Limitar a visibilidade do tráfego evitando que dados não pertinentes a determinada máquina estejam visíveis a outras é uma maneira simples e eficaz para diminuir a eficiência de um *sniffer*. O modo mais comum de implementar tal técnica é através da utilização de hardware especializado (como *switches*), configurações de redes onde haja separação entre as partes não relacionadas e utilização de rotas bem implementadas.

A tarefa de limitar a visibilidade do tráfego exige planejamento, hardware especializado e intenso trabalho de administração.

2.5.2 Utilização de Interfaces que Não Suportem Modo Promíscuo

Como já foi visto, a utilização de interfaces em modo promíscuo aumenta em muito o potencial de captação de um *sniffer*. Frente a isso, a utilização de interfaces que não operem nesse modo consiste numa boa maneira de limitar a utilidade de potenciais *sniffers* na rede, a não ser que sejam utilizados em roteadores, por exemplo, onde a utilização do modo promíscuo não é necessária.

Embora pareçam de grande utilidade, interfaces com essa característica não alcançaram sucesso quando colocadas no mercado. A principal razão para tal insucesso é o fato de que, além de fugir da especificação dos padrões do mercado há uma consequente perda de funcionalidade[5].

Existem contudo interfaces cujo *driver* é problemático e não funciona em modo promíscuo.

[5] Interfaces em modo promíscuo tem grande utilidade, como a criação de *bridges* [5] e a depuração de problemas na rede.

Esse fator é de pouca relevância uma vez que nada impede que um atacante crie e utilize seu próprio *driver* após conseguir privilégios de administrador em uma máquina, mas pode ser levado em conta em casos muito particulares, como a implementação de kernels monolíticos ou com camadas extras de segurança[6].

2.5.3 Utilização de Criptografia

A solução mais eficiente para o problema de acesso indevido a dados é torná-los ilegíveis ou inválidos para o atacante que os consiga capturar. Tal objetivo pode ser alcançado através da utilização de protocolos e canais criptografados – como túneis e VPNs [19] – e outras técnicas de criptografia, como extensamente discutido em [35]. É importante lembrar que embora seja um meio eficiente de garantir o sigilo das informações trafegadas, mesmo protocolos considerados seguros, se não bem implementados, podem ser quebrados com pouco ou nenhum esforço [1, 24].

Protocolos como SSL (Secure Socket Layer) [8, 12] e SSH (Secure SHell) [3] permitem o tunelamento de canais de comunicações de modo que todo o tráfego seja criptografado, e consistem em boas soluções para a implementação de redes seguras.

A utilização de protocolos não criptografados ainda é prática comum. Embora existam alternativas e soluções já há muito tempo disponíveis, o custo de implementação e manutenção adicional que estes acarretam os tornam proibitivos para aplicações em redes simples ou que tenham grande demanda de tráfego. Como mostrado em [2], implementar a utilização de canais criptografados utilizando SSL, pode exigir até o dobro de capacidade de processamento de um servidor ou cliente.

Uma breve lista de protocolos e suas principais características relacionadas à segurança do tráfego de informações, assim como as alternativas existentes é mostrada abaixo. Os detalhes sobre o funcionamento e arquitetura de cada protocolo não estão no escopo deste trabalho. Para isso recomendamos a consulta da documentação dos projetos que implementem tais protocolos ou seus respectivos RFCs [7].

- POP (Post Office Protocol) [26] e IMAP (Internet Message Access Protocol) [9]

  Protocolo para tráfego de correio eletrônico. É amplamente utilizado e tanto autenticação como dados são transmitidos em claro.
  Alternativas: Tunelamento sob SSL e utilização de mensagens criptografadas.

[6] Um exemplo é o projeto *Linux Intrusion Detection/Defense System* (LIDS), onde é criado um modelo de controle de acesso para o próprio kernel. Página do projeto: `http://www.lids.org`.

[7] RFCs, ou "Requests for Comments", são documentos criados com o intuido de criar padrões para a Internet. As transcrições podem ser encontradas em `www.rfc-editor.org`.

- SMTP (Simple Mail Transfer Protocol) [31]

  Protocolo de envio de correio eletrônico. Todo o tráfego é transmitdo em claro.
  Alternativa: Tunelamento sob SSL.

- Telnet [29]

  Um dos primeiros protocolos a ser criado para logins remotos, o telnet ainda hoje é utilizado e apresenta um grande risco uma vez que todo o tráfego (o que inclui nome de usuários e senhas) trafega em claro.
  Alternativa: SSH.

- NIS (Network Information Service) [41]

  Provê um conjunto de informações para o gerenciamento de usuários, máquinas e serviços de uma rede. Permite o livre acesso a nomes de usuários, senhas criptografadas e serviços disponíveis. É uma grande fonte de informação para atacantes e é passível de ataques de dicionário (força bruta).
  Alternativas: Kerberos, LDAP.

- HTTP (Hyper Text Transfer Protocol) [15]

  Dada sua popularidade, é amplamente utilizado, servindo como camada intermediária na implementação de inúmeros serviços a usuários. Nenhum tipo de criptografia é fornecido pelo protocolo, o que expõe todo o tráfego.
  Alternativa: HTTPS (HTTP + SSL)

- FTP (File Transfer Protocol) [30]

  Protocolo para transferência de arquivos. Inclui autenticação que, assim como todo o tráfego, é transmitida em claro.
  Alternativa: SSH

- Kerberos [21]

  Protocolo de autenticação que permite a reutilização de credenciais de login *("Single Sign On")*. A autenticação e a troca de chaves é feita de forma criptografada. Se não cuidadosamente implementado, é passível de ataques de sincronia e de dicionário (força bruta), como mostrado em [4].

- CVS (Concurrent Versions System) [10]

  Sistema de controle de versões amplamente utilizado para o gerenciamento de código fonte e documentação de projetos. Seu protocolo de comunicação nativo (*pserver*) não é criptografado, e o método de autenticação é extremamente simples.
  Alternativa: Tunelamento sob SSH

- IRC (Internet Relay Chat) [27]

  Protocolo para conversas via rede. Implementado sem grandes preocupações com segurança, transmite todo o tráfego em claro, e é utilizado como grande fonte de senhas e informações relevantes para atacantes, como alertado em [7]. Este protocolo deve ter sua utilização evitada quando houver a necessidade de transmissão de dados sensitivos.

O uso de canais de comunicação criptografados definitivamente se mostra como a melhor solução para o problema da visibilidade dos dados, já que os torna incompreensíveis a terceiros. Com o uso de criptografia, consegue-se anular boa parte da utilidade de um *sniffer* que esteja à procura de tráfego relevante. Porém a substituição de sistemas funcionais, a necessidade de maior capacidade de processamento e, de um modo geral, a configuração adicional associada à utilização de técnicas de criptografia – como a geração de certificados e chaves por parte do administrador – têm impedido sua ampla utilização.

2.5.4 Detecção

Mesmo em redes nas quais protocolos criptografados sejam utilizados e o tráfego seja bem delimitado, ainda é grande a quantidade de informações que um atacante munido de um *sniffer* consegue capturar. A topologia da rede, a versão dos softwares em execução, a carga e o número de usuários, além de diversos outros dados têm um valor substancial na formulação de ataques sofisticados.

A simples existência de um *sniffer* não autorizado em qualquer ponto da rede é um forte indício de que esta está sob a mira de atacantes ou usuários mal intencionados, independentemente da qualidade e quantidade dos dados que estes possam capturar.

Sendo assim, vê-se a necessidade da utilização de métodos que detectem a presença de *sniffers* e a ocorrência de incidentes em geral que sejam suspeitos. Tais métodos são discutidos no próximo capítulo.

Capítulo 3

Detecção de *sniffers*

Sniffers são em geral agentes passivos. Em outras palavras, eles raramente interferem no funcionamento da rede. Um *sniffer* ideal (no sentido de ser invisível aos usuários da rede) não injeta pacotes na rede, não responde a qualquer tipo de requisição e não precisa sequer ter um endereço da rede. Um exemplo de tal *sniffer* seria um dispositivo eletrônico qualquer capaz de capturar o tráfego diretamente do meio de transmissão sem a necessidade de pertencer formalmente à rede, trabalhando de maneira totalmente passiva. Detectar tal dispositivo seria uma tarefa de extrema dificuldade.

Felizmente, *sniffers* ideais como o citado acima são muito raros e, na prática, a principal característica visível de um *sniffer* é uma interface em modo promíscuo sem o aval do administrador da rede. É essa a característica geralmente procurada por um detector de *sniffers*.

Entre as diversas técnicas e ferramentas existentes para o cumprimento de tal tarefa, podemos destacar as relacionadas abaixo.

3.1 Detectores de Intrusão

Os detectores de intrusão *(Intrusion Detection Systems - IDS)*, tanto os destinados à redes *(Network Intrusion Detection Systems - NIDS)* como os destinados a uma única máquina *(Host Based Intrusion Detection Systems)* são caracterizados pela análise de tráfego e informações que evidenciem tentativas ou ocorrências de ataques, atuando como espécies de alarmes. IDSs são itens indispensáveis a redes nas quais haja um sério comprometimento com a questão da segurança.

Um detector de *sniffers* pode ser considerado uma espécie de NIDS (ou uma extensão destes), embora seu funcionamento e implementação, de um modo geral, sejam bastante diferentes.

3.2 Detecção Local

Na detecção local, o administrador do sistema precisa procurar por características de *sniffers* diretamente em cada máquina conectada à rede, sendo a principal delas a configuração da interface em modo promíscuo e certos processos em execução. Uma alternativa à verificação manual é um serviço que fique em execução[1] em cada máquina disparando um alarme quando alguma interface entrar em modo promíscuo ou tráfego de rede não endereçado à máquina em questão for recebido pelas camadas superiores da pilha de rede.

Apesar de ser um método determinístico, a detecção local tem sérias desvantagens:

Difícil implantação: é necessária a intervenção pessoal do administrador, ou de um software. A utilização de métodos de detecção local torna-se difícil em redes de grande extensão ou heterogêneas. Além disso, máquinas que não façam oficialmente parte da rede não seriam verificadas.

Não confiabilidade: o principal fator que reduz a confiabilidade de testes locais é o possível comprometimento da segurança da máquina invadida. Uma vez que tenha obtido privilégios de administrador na máquina, o atacante pode utilizar-se de vários artifícios para camuflar as respostas do sistema. Exemplos de tais artifícios são a substituição de utilitários do sistema, módulos e *drivers* do sistema operacional.

3.3 Detecção Remota

A detecção remota consiste na análise do tráfego da rede à procura de "assinaturas" típicas de *sniffers* ou de interfaces em modo promíscuo. Tais assinaturas possuem características decorrentes de peculiaridades do sistema operacional, do software em execução ou mesmo do hardware. Cabe a um detector o papel de explorar essas peculiaridades, muitas vezes enviando tráfego que dispare um determinado comportamento na máquina remota. Dentre essas características, uma possível classificação dos testes remotos é a de *ativos* e *passivos*, no sentido de alterarem ou não o estado da rede através da injeção de pacotes.

Dado que *sniffers* são geralmente programas executados a partir de máquinas de uso geral e assumindo que tais máquinas têm uma implementação de uma pilha TCP/IP e outros protocolos e serviços disponíveis à rede, pode-se desenvolver testes remotos que explorem suas características.

[1] Uma possibilidade é a utilização do protocolo SNMP [14].

Como a definição do comportamento de uma pilha TCP/IP é bastante flexível e não aborda todas as combinações possíveis de pacotes e opções (*flags*), ela é implementada de inúmeras maneiras diferentes. É possível, por exemplo, a detecção da versão do sistema operacional e diversas outras características de uma máquina através de uma técnica conhecida como "TCP/IP Fingerprinting"[2], através da qual são analisadas respostas a determinados tipos de requisições e reações a comportamentos inesperados [18].

A maioria dos testes remotos baseiam-se na exploração de características de determinadas implementações da pilha TCP/IP e do próprio *driver* de rede, especialmente as mais comuns em sistemas operacionais atuais.

As próximas subseções contêm uma explicação detalhada dos métodos conhecidos [20] para a detecção remota de *sniffers* ou de interfaces trabalhando em modo promíscuo.

3.3.1 Requisição ICMP com Falso Endereço Físico

Um dos testes de detecção remota mais conhecidos consiste no envio de um pacote do tipo ICMP ECHO REQUEST utilizando um endereço destino de hardware adulterado de modo a não ser normalmente aceito pela máquina suspeita, a não ser que esta se encontre em modo promíscuo. Uma resposta a esse tipo de requisição indica que a máquina testada está recebendo tráfego através de uma interface de rede em modo promíscuo.

Excluindo-se o endereço de *broadcast* e os endereços de *multicast* [11] configurados para serem aceitos, uma interface de rede trabalhando em modo normal só deveria aceitar quadros enviados com seu próprio endereço de hardware no campo de destino.

Uma interface trabalhando em modo promíscuo passa todo quadro recebido às camadas superiores da pilha de rede. Por questões de modularidade e eficiência, tais camadas fazem poucas verificações nos endereços de hardware dos quadros repassados. É enviando pacotes com endereços de hardware válidos dentro destas verificações, porém fora do conjunto de endereços normalmente aceitos pela interface (*broadcast*, *multicast* e o próprio endereço da interface) que se faz possível a realização deste teste.

Tomando como exemplo uma implementação do módulo de rede do kernel Linux versão 2.4, quando um endereço de hardware não coincide com o endereço da própria interface, seu primeiro octeto é testado contra o valor *0xff*. Caso este teste seja verdadeiro, assume-se que foi recebido um quadro do tipo *broadcast* ou *multicast*. Não sendo do tipo *broadcast*[3], este mesmo octeto é testado novamente; se for ímpar, assume-se que o quadro é do tipo *multicast*, caso contrário ele é sumariamente descartado. Tal comportamento pode ser evidenciado nas funções

[2] A conhecida ferramenta "nmap" em sua versão 3.0 é capaz de detectar mais de 700 diferentes implementações utilizando tal técnica. `http://www.insecure.org/`.

[3] O valor para *broadcast* Ethernet é `{0xff,0xff,0xff,0xff,0xff,0xff}`.

ip_rcv(), eth_type_trans() e *ether_setup()* da camada de rede do kernel Linux[4] nas versões 2.2, 2.4 e 2.5.

Este teste também pode funcionar em algumas redes comutadas, uma vez que existem *switches* que ao receberem pela primeira vez um pacote com endereço desconhecido o repassam a todos os segmentos da rede.

Embora comumente se utilizem pacotes do protocolo ICMP para a realização deste teste, este também pode ser implementado utilizando-se outros protocolos, como HTTP ou FTP.

No restante deste trabalho referimo-nos ao teste de requisições de ICMP com falso endereço físico simplesmente como **teste ICMP**.

3.3.2 Requisição ARP com Falso Endereço Físico

De forma similar ao teste visto anteriormente, este teste utiliza-se de requisições do protocolo ARP (Address Resolution Protocol) [28], que é o protocolo utilizado para converter endereços IP para endereços físicos em redes Ethernet. Neste teste, é feita uma requisição com endereço físico de destino inválido e aguarda-se uma resposta por parte da máquina suspeita.

Explorando a pouca variação nas implementações do sistema de resolução de endereços de determinados sistemas operacionais – até mesmo devida à simplicidade deste protocolo –, este teste mostra-se com um escopo bastante amplo, funcionando, por exemplo, na detecção de máquinas com ambientes Windows, Linux e BSD.

Máquinas geralmente possuem *cache* de ARP, tornando possível usar uma variação deste teste enviando anúncios ARP com endereço de hardware inválido no intuito de que máquinas com interfaces em modo promíscuo guardem no cache a tupla <*IP, endereço físico*> recebida através desses anúncios inválidos. Nesse cenário é feita uma requisição qualquer em modo *broadcast* na rede (como uma requisição de ECHO ICMP) e aguarda-se que alguma máquina (ou máquinas) responda a tal requisição sem fazer uma consulta ARP, o que indicaria a presença de uma interface atuando em modo promíscuo.

Um cuidado deve ser tomado com a ocorrência de falsos positivos, uma vez que se houver tráfego legítimo entre as duas máquinas, estas eventualmente precisarão trocar requisições e respostas ARP. Como não é possível identificar a partir de qual requisição veio uma determinada resposta, o teste pode erroneamente reportar que a máquina está em modo promíscuo.

No restante deste trabalho referimo-nos ao teste de requisições de ARP com falso endereço físico simplesmente como **teste ARP**.

[4]Os fontes do kernel Linux normalmente podem ser encontrados nos diretórios abaixo de */usr/src/linux* em distribuições Linux padronizadas, ou diretamente de `http://www.kernel.org`.

3.3.3 DNS Reverso

Para um atacante ou até mesmo um administrador, é muito mais fácil verificar os dados coletados por um sniffer analisando nomes de host ao invés de confusos endereços IP. É principalmente por causa desta razão que muitos sniffers optam por trocar os endereços IP coletados por nomes de host utilizando a funcionalidade de resolução reversa de nomes do DNS.

Tendo acesso físico ao caminho de rede entre uma máquina suspeita e o servidor de nomes utilizado por ela, é possível arquitetar um teste de detecção remota que descubra um sniffer através da verificação da utilização desta funcionalidade (resolução reversa) por parte do sniffer. Para isso, é simulada uma conexão entre uma máquina interna e outra externa à rede testada, com o detalhe de que o endereço IP da máquina externa é escolhido de forma absurda, ou seja, ou ele simplesmente não existe alocado a algum domínio ou é um endereço cuja probabilidade de utilização pelas máquinas da rede testada seja quase nula. A partir disto, espera-se que alguma máquina pertencente ao segmento de rede em que a conexão foi simulada tente acessar o servidor DNS em busca de informações sobre este endereço falso.

O endereço IP utilizado deve ser escolhido com cautela. Este não deve ser comum à rede testada, uma vez que requisições de resolução legítimas geradas pelas máquinas testadas poderiam criar falsos positivos. Além disso, o tráfego simulado deve utilizar um protocolo que seja interessante ao *sniffer* procurado, evitando ser descartado por um filtro.

Um detalhe a ser considerado nesse teste é que ele pode deixar de detectar *sniffers* após um determinado número de execuções. Isso se dá devido ao fato de que o *sniffer* pode ”desistir”de tentar resolver o endereço IP após um determinado número de tentativas[5]. Uma vez que o *sniffer* pode ficar em execução por um longo período de tempo, recomenda-se que o teste seja feito com endereços IPs diferentes entre execuções consecutivas.

No restante deste trabalho referimo-nos ao teste de requisições de DNS reverso simplesmente como **teste DNS**.

3.3.4 Latência

Um *sniffer* pode consumir recursos computacionais consideráveis de uma máquina para fazer coleta e análise de tráfego. Se a interface estiver em modo promíscuo e a quantidade de tráfego for substancialmente grande, a responsividade da máquina pode diminuir de forma perceptível. Uma maneira não determinística mas eficiente de detectar-se a presença de *sniffers* é através da medida de variação de tal responsividade.

[5]Alguns sistemas operacionais podem implementar um sistema de *cache* para as informações obtidas através de requisições DNS.

O objetivo do teste de latência é gerar uma condição de negação de serviço (*Denial of Service - DoS*) na máquina alvo através da utilização de algum tipo de tráfego que sobrecarregue apenas máquinas que estejam com um *sniffer* em execução. A escolha de tal tráfego deve levar em conta inúmeras variáveis do sistema operacional, do equipamento utilizado e do *sniffer* em execução.

Esse teste pode ser feito através da medida do tempo de resposta e do número de requisições respondidas com a utilização de requisições ECHO do protocolo ICMP ou através da mensuração da responsividade de qualquer serviço em execução na máquina suspeita.

Como insere uma grande quantidade de tráfego na rede, ele deve ser utilizado com muito cuidado, pois pode interferir no desempenho e funcionamento de serviços daquela. Essa característica faz com que sua utilização seja mais adequada quando já existe a suspeita da execução de um *sniffer* em alguma máquina. Esse teste deve ser ajustado para a rede em questão e seus resultados analisados com cautela, uma vez que são subjetivos e não determinísticos. A implementação do serviço usado para medição, a situação da rede no momento e vários outros fatores podem afetar o tempo de resposta mensurado.

Abaixo temos uma lista com alguns dos mais importantes fatores que devem ser levados em conta na geração da inundação:

- Pacotes de protocolos complexos

  Alguns *sniffers* analisam os campos internos de pacotes de determinados protocolos para a captura de informações específicas. Entre esses protocolos, podemos citar os que podem conter informações sensitivas como senhas e nomes de usuários e os que têm estrutura complexa, como os pacotes utilizados pelo protocolo de resolução de nomes (DNS). Tais protocolos precisam ser do interesse do atacante para não serem descartados inicialmente no filtro de pacotes do *sniffer*.

- Pacotes pequenos

  Como o *sniffer* precisa fazer análise do tráfego baseado em pacotes, os cabeçalhos de todos estes precisam ser abertos. Sendo assim, utilizar um grande número de pacotes pequenos em geral é vantajoso em relação à utilização de poucos pacotes grandes.

- Truques com o protocolo TCP

  A utilização de algumas opções de pacotes TCP e a exploração de algumas características deste protocolo podem esgotar os recursos de um máquina, como descrito em [36, 25].

- Inundação distribuída

  Técnicas de negação de serviço distribuída *(Distributed Denial of Service - DDoS)* podem ser utilizadas para fazer com que a máquina alvo do teste tenha sua performance (processamento) seriamente comprometida. Nesse cenário, diversas máquinas são utilizadas com

o intuito de sobrecarregar a máquina alvo [39]. No caso da detecção de *sniffers*, conexões com falsos endereços de hardware podem ser utilizadas, sobrecarregando a máquina apenas se esta estiver em modo promíscuo.

- Capacidade de processamento

  É importante levar em conta a capacidade de processamento tanto da máquina alvo como da máquina geradora da inundação. Enquanto que a tarefa de geração de pacotes pode consumir uma substancial quantidade de recursos da máquina, se o alvo tiver grande capaciade de processamento a inundação pode não ter o efeito esperado para este teste. Além disso, a capacidade da rede deve ser levada em consideração, pois se esta não for capaz de suportar a quantidade de tráfego necessária, a inundação não terá o efeito desejado.

Outros estudos sobre ataques de negação de serviço, assim como técnicas para sua contenção podem ser encontrados em [6, 34, 23].

Como no teste de requisições DNS, é possível implementar esse teste de maneira a testar a existência de *sniffers* em todas as máquinas da rede ao mesmo tempo, uma vez que a inundação de pacotes é visível a todas elas.

3.3.5 Armadilha *(Honey Pot)*

O uso de armadilhas é uma técnica amplamente usada na detecção e estudo de ataques de diversos tipos. Uma armadilha, comumente chamada de “Honey Pot”, consiste na utilização de “iscas” – geralmente máquinas em ambientes bem controlados – para serem invadidas e exploradas por atacantes [13]. A partir das informações coletadas em tais ambientes é possível conhecer melhor o atacante, suas técnicas e em muitos casos chegar à sua identificação.

No caso da detecção de *sniffers*, pode-se fornecer falsas senhas e informações através de conexões forjadas e então monitorar seu uso na rede, podendo ser utilizados quaisquer protocolos que venham causar interesse a um atacante. O exemplo mais comum é o uso de contas de e-mail (POP/IMAP) cuja autenticação seja feita em texto claro.

Uma possível implementação pode agir como um *sniffer* procurando pelo uso dos dados falsos ou expandir a armadilha a outros níveis, criando falsos ambientes também para os serviços cujas informações falsas foram disponibilizadas.

A utilização deste tipo de técnica é bastante eficiente pois pode conseguir detectar até mesmo um *sniffer* totalmente passivo, além de se mostrar muito útil no estudo de hábitos do atacante. Seu grande problema é que a resposta para o teste pode demorar consideravelmente e sua implantação pode exigir alterações nos serviços da rede. Alguns exemplos destas alterações

podem ser a implantação de serviços de rede que sejam passivos à atuação dos atacantes e a simulação de ambientes corporativos que atraiam a atenção e previnam a desconfiança por parte dos atacantes.

3.3.6 Detecção de Inundação de Respostas ARP

sniffers em redes comutadas geralmente se fazem passar por outra máquina para receber o tráfego e fazer seu roteamento (veja seção 2.4.2). Para isso, uma das técnicas freqüentemente empregadas consiste no envio de anúncios ARP em excesso para enganar outras máquinas da rede. Tal excesso, comumente chamado de inundação *(flooding)*, constitui uma evidência de um *sniffer* ou atacante em atuação.

Para a implementação de tal teste de maneira confiável, é necessária a definição de um limiar a partir do qual o número de anúncios é considerado excessivo. Este deve ser ajustado conforme as características da rede e dos *sniffers* em questão.

3.4 Confiabilidade dos Testes

O principal problema com os métodos de detecção remotos é que eles, em sua maioria, podem ser evadidos por *sniffers* ou atacantes que conheçam a metodologia utilizada. Um atacante pode alterar o comportamento do SO ou implementar "técnicas de detecção das técnicas de detecção", criando *sniffers* que reajam aos testes de maneira a não serem detectados. É importante notar que tal evasão não é trivial e não se tem notícia de *sniffers* que tenham implementado tais técnicas até o momento.

Além disso, a grande variedade nas implementações da pilha TCP/IP se mostra um desafio para a implementação de testes portáveis, que venham a detectar *sniffers* ou interfaces em modo promíscuo em uma ampla variedade de arquiteturas e sistemas operacionais.

Devido à característica inerentemente passiva dos *sniffers* e a pressuposição de que uma máquina invadida não é confiável, a tarefa de detecção é complexa. A melhor maneira de proteger os dados em trânsito ainda é o uso da criptografia juntamente com a limitação na disponibilização do tráfego. A utilização de técnicas de detecção porém consiste numa das muitas armas disponíveis aos administradores de redes na luta pela segurança dos dados e, na guerra contra os atacantes, todas as armas têm sua utilização valorizada nas várias frentes de batalha.

Capítulo 4

Implementação

A escassez de programas para detecção de *sniffers* em redes, principalmente gratuitos e de qualidade, além do interesse pela área de segurança computacional motivou a implementação de um aplicativo para detecção remota de *sniffers*.

O projeto, batizado de *sniffdet*, foi dividido em duas partes, a saber, uma biblioteca contendo os testes de detecção implementados, chamada *libsniffdet*, e uma aplicação para utilizar e testar esta biblioteca. A vantagem desta divisão está no fato de que uma vez definida a API da biblioteca, ambas as partes poderiam ser desenvolvidas simultaneamente até o estágio de depuração, otimizando o tempo usado para implementação.

O restante deste capítulo mostra os princípios, as qualidades, orientações e preocupações relevados durante o processo de desenvolvimento do projeto *sniffdet*.

4.1 Preocupações do Projeto

O projeto começa com a definição da interface de seu componente mais importante, a biblioteca *libsniffdet*. Sabe-se, a princípio, que preocupações comuns em desenvolvimento de software tais como eficiência, eficácia, modularidade e manutenibilidade devem ser somadas a outros itens específicos de acordo com sua aplicação e utilização. Alguns destes itens apresentam-se ressaltados abaixo.

Segurança como se trata de um aplicativo que lida com softwares e dados potencialmente maliciosos, a segurança dentro do ambiente de execução é primordial, sendo levada em consideração durante todo o processo de desenvolvimento.

Extensibilidade dada a dinamicidade da área de detecção de *sniffers*, é esperado que novos testes e variações dos já existentes sejam criados com o passar do tempo, devendo ser incorporados à biblioteca.

Flexibilidade *Sniffers* podem alterar seu comportamento de modo a evitar sua detecção, principalmente se os testes de detecção possuírem algum tipo de comportamento que os denuncie, ou seja, algum tipo de "assinatura" que os caracterize. Pensando nisso, a utilização de valores constantes é evitada e são disponibilizados vários artifícios para se configurar os atributos dos testes de detecção.

Usabilidade Pouco vale um sistema para auxiliar administradores de rede se somente uns poucos forem capazes de utilizá-lo. O software é acompanhado de extenso material explicando seu funcionamento e sua utilização e é adotada a utilização de valores padrão para alguns dos parâmetros omitidos pelo usuário, tornando seu uso mais prático.

Responsividade É importante fornecer um mecanismo de comunicação entre a biblioteca e a aplicação que funcione durante a execução dos testes, permitindo uma melhor interação entre os dois. Este mecanismo evita a impressão de "congelamento", principalmente naqueles testes que exigem bastante tempo para realizarem medições mais precisas.

4.2 Documentação

A preocupação com a utilização do sistema fomentou a criação de vários documentos que descrevem a utilização da *libsniffdet* e sua aplicação, bem como seu funcionamento interno, convenções de suas *APIs* e preocupações na utilização. Além destes textos específicos, existem páginas de manual (*manpages*), compilações de perguntas mais freqüentes (*faqs*) e a página oficial do projeto, disponível em `http://sniffdet.sourceforge.net`. Como é um projeto concebido para ser utilizado e receber contribuições da comunidade mundial de desenvolvimento de software livre, existem versões da documentação em português e inglês.

4.3 Licença e Distribuição

Todo o código fonte e documentação deste projeto estão sob a licença *GPL (General Public License)* versão 2 da GNU [17]. A GPL é uma licença de software que garante a livre distribuição e alteração deste contanto que os direitos autorais sejam mantidos e a licença não revogada.

O projeto *sniffdet* está sendo distribuído pela Internet e conta com toda a infra-estrutura necessária para receber contribuições da comunidade do software livre como novas funcionalidades, indicações de erros, sugestões, correções e discussões em geral a respeito do projeto.

O material produzido pode ser obtido através da página do projeto ou da utilização de um cliente CVS. A página do projeto está localizada em `http://sniffdet.sourceforge.net`. Para se obter o material diretamente do repositório de desenvolvimento (códigos fonte e documentação) basta usar um cliente CVS com `:pserver:anonymous@abrigo.dyndns.org:/sniffdet` como raiz.

O projeto está dividido em três módulos no repositório, a saber:

- **docs:** documentação gerada durante a execução do projeto;
- **sndet:** código fonte da *libsniffdet* e aplicação;
- **web:** estrutura e documentos da página do projeto.

Na página web são encontradas as últimas versões oficiais, correções para problemas sérios e pacotes binários em formato RPM.

4.4 Ambientação

Tanto a biblioteca quanto a aplicação foram implementados usando linguagem C a partir de uma plataforma operacional Linux. Dada a popularidade e disponibilidades das redes padrão Ethernet, este foi escolhido para a implementação e testes do projeto.

Foram utilizadas diversas ferramentas e bibliotecas auxiliares para o desenvolvimento do mesmo, sendo que as mais relevantes são:

- libpthread

  Biblioteca POSIX de gerenciamento de *threads*.

- glibc 2.2

  Biblioteca GNU padrão C.

- gcc (GNU C Compiler)

  Compilador GNU C.

- autoconf

  Utilitário para automação de configuração do sistema.

- libnet

  Biblioteca para criação de pacotes de rede.

- libpcap

  Biblioteca para captura de pacotes de rede.

- CVS (Concurrent Versions System)

  Sistema de gerenciamento para acesso concorrente a repositórios de código centralizados.

- GNU make

  Utilitário de construção automática de projetos.

- ElectricFence

  Wrapper da biblioteca padrão de alocação de memória. Útil para a fase de depuração, pois ajuda a encontrar vazamentos de memória e acessos a endereços inválidos.

- gdb (GNU Debugger)

  Depurador de processos GNU.

- Ethereal

  Sniffer com recursos de visualização de pacotes capturados, inclusive descrição dos protocolos utilizados e verificação de *payload*. Essencial para verificar a adequação dos pacotes gerados pela *libsniffdet* e a captura correta dos mesmos nas máquinas testadas.

- tcpdump

  Sniffer bastante flexível que funciona em interface texto.

Todos os itens acima são gratuitos, de livre distribuição e disponibilizados em várias plataformas operacionais diferentes. Juntando este fato à preocupação com a documentação, a adoção do padrão ANSI para linguagem C e a utilização de várias bibliotecas padrão POSIX, espera-se que o porte deste projeto para outras plataformas torne-se uma tarefa bem simplificada. Atualmente, ele já compila e é executado num sistema FreeBSD, sendo necessárias apenas algumas adaptações no processo de compilação.

4.5 Biblioteca

Bibliotecas são componentes de sistemas que agrupam funções e definições de propósito comum em objetos disponibilizados de maneira centralizada. Sua finalidade é agregar funcionalidades a programas em tempo de execução de acordo com a demanda, num processo chamado de

ligação dinâmica. A vantagem está no fato de que o código existente na biblioteca não precisa ser replicado em cada processo com interesse em utilizá-lo, economizando memória e o tempo de processamento levado para carregá-la.

Outra facilidade na utilização de bibliotecas é que as mesmas podem ser modificadas de forma transparente aos usuários, permitindo sua expansão e correção sem exigir a recompilação dos aplicativos que as usam. Para que este procedimento possa ocorrer sem problemas, deve-se seguir algumas regras como, por exemplo, as citadas em [42] e mencionadas abaixo.

1. Não trocar o comportamento de funções entre versões;
2. Não modificar o layout ou atributos de dados exportados que não são alocados somente internamente na biblioteca;
3. Não remover funções exportadas;
4. Não modificar a interface de uma função exportada;
5. Evitar inserir novos membros em estruturas em uma posição diferente da última;

O procedimento de ligação dinâmica trata, basicamente, da exportação de símbolos e as restrições acima garantem a compatibilidade entre diferentes versões de uma biblioteca, ou melhor dizendo, a compatibilidade binária das mesmas.

Todos estas qualidades juntas da possibilidade de inserção de novos testes de detecção de *sniffers* num futuro próximo justificam a escolha de uma biblioteca como residência dos testes implementados.

4.5.1 Arquitetura Geral

A arquitetura geral da biblioteca está representada pela figura 4.1. Como pode ser visto, a biblioteca está dividida em módulos, um para cada teste de detecção implementado. Os testes de detecção existentes na versão atual são o *ICMP*, *ARP*, *DNS* e *de Latência*. Existe um módulo especial, o módulo *helpers*, que agrega várias funções de propósito geral tais como tradutores de nomes e geradores de números aleatórios, podendo ser utilizado tanto pelos outros módulos quanto pela aplicação. A comunicação da biblioteca com a aplicação é feita através da ativação com passagem de parâmetros e do uso de funções de *callback*. O fluxo comum de utilização de suas funções segue abaixo.

1. Inicializar a estrutura de controle da interface de rede;
2. Chamar a função de teste requerida, passando todos os argumentos obrigatórios e, na medida do interesse, os opcionais;

3. Tratar, opcionalmente, os dados enviados pela callback;
4. Interpretar os resultados;
5. Desalocar a estrutura de controle da interface de rede.

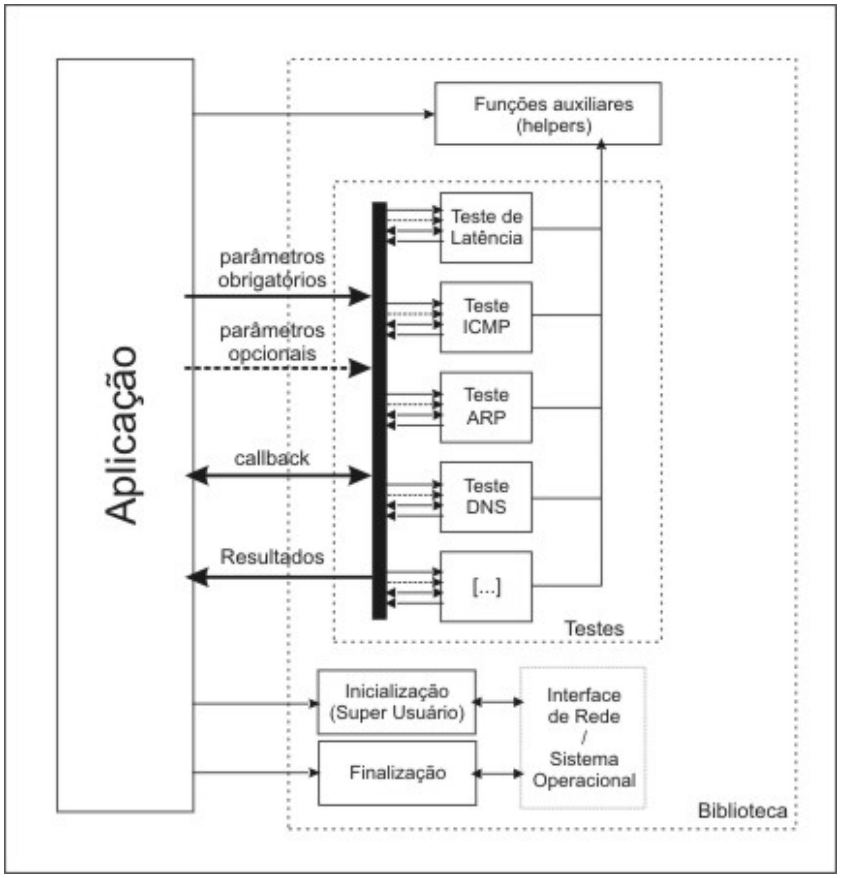

Figura 4.1: *Arquitetura geral da biblioteca.*

Dentre estes passos, o único que precisa ocorrer com privilégios de administrador é o primeiro, pois a interface precisa ser inicializada sem restrições de utilização. Portanto, a aplicação tem a opção de efetuar a restrição de privilégios - trocar sua identificação de usuário para outro diferente do administrador - após este passo inicial, contribuindo para uma execução mais segura dos testes.

Pensando na praticidade de sua utilização, a biblioteca incorpora o conceito de classificação de argumentos, dividindo-os em 2 tipos: obrigatórios e opcionais. Os argumentos minimamente necessários para a correta execução de um teste são ditos obrigatórios e sua omissão gera um aviso em tempo de execução e a pronta parada do teste. A omissão dos argumentos opcionais faz com que a biblioteca substitua seus valores internamente por padrões que se mostrem adequados. Os valores escolhidos para esta substituição são discutidos logo a seguir.

4.5.2 Funcionamento

Todos os testes de detecção até agora implementados funcionam com a utilização de múltiplas *threads*. Os testes são do tipo de detecção remota ativa, que injetam pacotes na rede e coletam pacotes ou impressões (no caso do teste de latência). Podemos generalizar seu modelo de acordo com a figura 4.2.

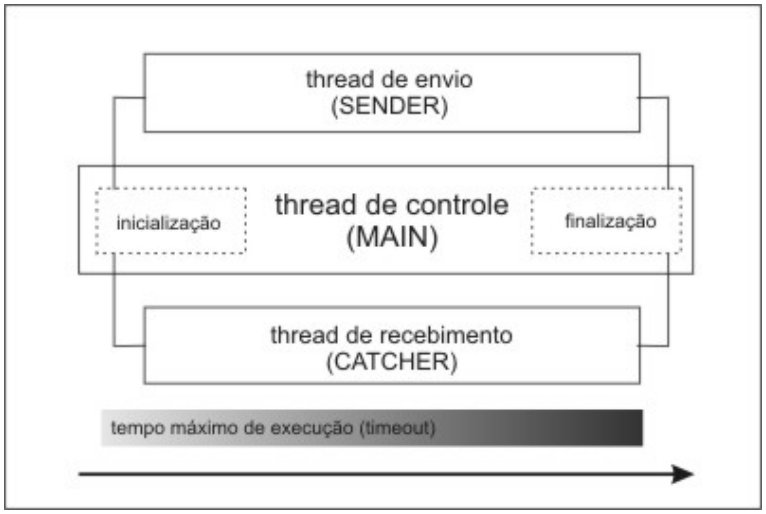

Figura 4.2: *Funcionamento das threads dos testes de detecção.*

São três as *threads* responsáveis pelos testes. Uma delas - *main* - é responsável pelo controle das outras duas *threads*, incluindo a criação, destruição, inicialização, desalocação de suas estruturas de controle e monitoração do tempo máximo de execução, representado pela barra de timeout. As outras duas *threads* são responsáveis, respectivamente, pelo envio de estímulos à rede *(sender)* e pela percepção de respostas que caracterizem a utilização de *sniffers* na rede testada *(catcher)*, sendo estas as características inerentes aos testes de detecção ativos.

4.5.3 Responsividade

Alguns testes de detecção podem precisar de vários minutos de funcionamento para realizar uma verificação significativa do ambiente a ser testado. Durante este tempo, é importante que haja algum indício do que o programa está fazendo atualmente a fim de se evitar a impressão de "congelamento" por parte do usuário. Por outro lado, pode ocorrer a necessidade de se cancelar um teste em execução ao invés de esperar pelo seu término. Estas características são ainda mais necessárias em ambientes com interfaces gráficas.

A preocupação com a responsividade do sistema foi satisfeita através da implementação de um mecanismo de comunicação entre a biblioteca e a aplicação. A partir da implementação e freqüente ativação de uma função compatível com o protótipo abaixo, é possível obter notificações sobre o andamento da execução dos testes de detecção ao mesmo tempo em que

a aplicação pode requisitar o cancelamento de um procedimento em andamento. Este tipo de função também é chamada de *função de callback*.

```
int (*user_callback)(struct test_status *status, int msg_type,
     char *msg);
```

O primeiro argumento simplesmente mostra dados estatísticos, como a porcentagem de execução do teste ativado e a quantidade de bytes recebidos e enviados. O segundo argumento identifica o tipo da mensagem. O terceiro traz uma mensagem a respeito do evento que ativou a chamada desta callback quando necessária e o valor de retorno indica a requisição de cancelamento do teste em execução.

São seis os tipos de mensagens passadas à aplicação pela biblioteca:

- **RUNNING**: serve para indicar o correto funcionamento do teste de detecção ativado e afastar a impressão de "congelamento";
- **NOTIFICATION**: vem acompanhado de informações corriqueiras sobre o funcionamento do teste, como seu instante de início ou término;
- **ERROR**: indica uma condição crítica e iminente cancelamento do teste em execução, tal como erros em chamadas de sistema ou falta de permissão para manipular a interface;
- **WARNING**: mostra alertas relevantes, porém, que não levam ao cancelamento do teste;
- **DETECTION**: indica a detecção de um *sniffer*;
- **ENDING**: avisa sobre o fim da execução do teste;

Outro fator importante sobre a utilização de callbacks é que como sua implementação ocorre no espaço de código da aplicação, cada programa usuário pode desenvolvê-la de acordo com sua necessidade, podendo criar mecanismos diferenciados de registro de acordo com o tipo de mensagem recebida, assim como ativar sistemas externos a partir dela. Um exemplo disso seria o envio de e-mail ao administrador da rede testada quando da detecção de um *sniffer* na mesma.

4.5.4 Resultado dos Testes de Detecção

Algumas aplicações podem implementar o conceito de bateria de testes. Portanto, para facilitar a classificação e posterior visualização dos resultados, foi criada uma estrutura comum de retorno dos testes que identifica o teste realizado através de seu código (enumeração), nome, descrição, tempo de início e fim de sua execução, além de uma indicação sobre sua validade (correta execução) e algumas informações estatísticas. Esta estrutura é mostrada abaixo:

```
struct test_info {
    enum test_code code;
    int valid;
    char *test_name;
    char *test_short_desc;
    time_t time_start;
    time_t time_fini;
    unsigned int b_sent;
    unsigned int b_recvd;
    unsigned int pkts_sent;
    unsigned int pkts_recvd;
    union {
        struct icmptest_result icmp;
        struct arptest_result arp;
        struct dnstest_result dns;
        struct latencytest_result latency;
    } test;
};
```

Cada teste possui uma estrutura de retorno específica de acordo com suas características. Cada uma delas será explicada nas subseções à seguir, junto das implicações práticas de utilização de cada teste.

4.5.5 Teste ICMP

O teste de detecção usando mensagens ICMP apresenta o seguinte protótipo.

```
int sndet_icmptest(char *host,
    struct sndet_device *device,
    unsigned int tmout,
    unsigned int tries,
    unsigned int send_interval,
    user_callback callback,
    struct test_info *result,
    char *fakehwaddr
    );
```

Os argumentos obrigatórios são a estrutura de controle da interface de rede (*device*) e o endereço da máquina a ser testada (*host*).

A flexibilidade e a descaracterização de assinaturas necessárias para se evitar a contra-atuação por parte de *sniffers* mais avançados são garantidas pela variação do valor dos argumentos de intervalo de envio de pacotes (*send interval*) e endereço MAC utilizado como destino (*fakehwaddr*). A omissão destes valores faz com que sejam internamente substituídos por um segundo de intervalo de envio de pacotes - valor comum para aplicativos de diagnóstico de

rede, tais como o ping - e pelo endereço `{0xff,0x00,0x00,0x00,0x00,0x00}`, conforme sugestão explicada em 3.3.1.

Seu funcionamento dentro do modelo genérico apresentado consiste na utilização de uma *thread* para enviar as requisições ICMP falsas e outra *thread* para capturar respostas enviadas pela máquina alvo.

Devido à natureza determinística deste teste, sua parte específica da estrutura de retorno contém somente uma argumento que diz se o alvo testado foi flagrado ou não com um *sniffer* em execução.

4.5.6 Teste ARP

O teste ARP apresenta o seguinte protótipo e tem um comportamento similar ao do teste ICMP:

```
int sndet_arptest(char *host,
    struct sndet_device *device,
    unsigned int tmout,
    unsigned int tries,
    unsigned int send_interval,
    user_callback callback,
    struct test_info *result,
    char *fakehwaddr
    );
```

De acordo com sua semelhança com o teste ICMP, tem os mesmos argumentos obrigatórios, assim como valores padrão, funcionamento geral e estrutura de retorno.

4.5.7 Teste DNS

O teste DNS apresenta o seguinte protótipo:

```
int sndet_dnstest(char *host,
    struct sndet_device *device,
    unsigned int tmout,
    unsigned int tries,
    unsigned int send_interval,
    user_callback callback,
    struct test_info *info,
    char *fake_ipaddr,
    char *fake_hwaddr,
    ushort dport, ushort sport,
    char *payload,
    short int payload_len
    );
```

Tem os mesmos argumentos obrigatórios dos testes anteriores, porém, a flexibilidade e descaracterização de assinaturas é mais elaborada. São usados os argumentos de tempo de envio de pacotes (*send interval*) e outros mais para simular um pacote de comunicação comum, através da parametrização dos endereços IP de destino e origem, das portas de destino e origem e de um payload opcional.

Os seguintes valores são usados quando da omissão dos argumentos opcionais:

- Um segundo como intervalo de envio de pacotes;
- Porta de origem 23 (Telnet) e destino 1150 (nenhum serviço especial);
- Endereço MAC destino `{0x44,0x44,0x44,0x44,0x11,0xff}`;
- Endereço IP destino `10.0.0.21`.

Tanto o endereço MAC quando o IP são endereços inválidos na rede testada

O funcionamento da *thread* de leitura tem um cuidado especial no que diz respeito à leitura do pacote de consulta DNS. Como o próprio dado contido na requisição DNS orienta sobre a formação dos nomes representados no pacote, um *sniffer* pode construir um pacote mal-formado com o intuito de causar uma falha no teste de detecção através do acesso à memória indevido. Este problema é contornado respeitando sempre o tamanho total do pacote capturado.

A estrutura de resposta deste teste é semelhante a do ICMP e ARP, com somente uma indicação da localização ou não do *sniffer* na máquina testada.

4.5.8 Teste de Latência

O teste de latência apresenta o seguinte protótipo:

```
int sndet_latencytest_pktflood(char *host,
    struct sndet_device *device,
    unsigned int tmout,
    unsigned int probe_interval,
    user_callback callback,
    struct test_info *info,
    struct custom_info *bogus_pkt
    );
```

Os argumentos obrigatórios são o nome da máquina alvo (*host*) e a estrutura de controle da interface de rede (*device*). Para a descaracterização de assinaturas, têm-se a parametrização do intervalo de verificação de latência e a estrutura de construção de pacote (*bogus pkt*), apresentada logo abaixo.

```
struct custom_info {
    int values_set;

    // ETH
    u_char dmac[6];
    u_char smac[6];

    // IP
    uint id;
    uint timestamp;
    u_char ttl;
    ulong dest_ip;
    ulong source_ip;

    // TCP/UDP
    short protocol; // udp/tcp/icmp
    int flags; // header flags
    uint seq;
    uint ack;
    ushort winsize;
    short dport;
    short sport;
    u_char *payload;
    short payload_len; // mandatory if payload is used
};
```

Com todos estes parâmetros, é possível criar vários tipos de pacotes diferentes para a inundação da rede. Como discutido no capítulo anterior, o importante é fazer com que o *sniffer* gaste o máximo de tempo com o processamento destes pacotes.

A omissão do argumento `bogus_pkt` faz com que a biblioteca o susbstitua internamente por um pacote telnet com o flag SYN ligado, simulando um início de conexão.

Devido à natureza não-determinística deste teste, a estrutura de retorno restringe-se a disponibilizar os dados colhidos durante o estágio de tráfego normal e de inundação da rede para que, através da comparação e de uma certa subjetividade, o usuário decida sobre a possibilidade de existência do *sniffer*. Os dados disponibilizados são o tempo médio de resposta na rede com tráfego normal e os tempos mínimo, médio e máximo e o número de pacotes enviados e perdidos na rede sobrecarregada.

Em futuras versões deste teste, pretende-se aprimorar o cálculo do tempo médio de resposta através do uso de desvio padrão ou outras técnicas estatísticas. A utilização de vários tipos de pacotes para inundação também é requerida, pois quanto mais se explorar a pilha da máquina alvo, maior será o tempo gasto por ela para processar os pacotes enviados, fornecendo uma medida melhor da latência incorrida em máquinas que rodam com sua interface em modo promíscuo.

4.6 Aplicação

O principal objetivo da aplicação é fornecer um modo simples e flexível para se testar a biblioteca. Além de cumprir esta tarefa, a aplicação propiciou uma melhor visão sobre a utilização dos testes de detecção. Durante sua codificação e testes, houve um amadurecimento da biblioteca, que teve uma melhor definição de que valores utilizar por padrão e como se comunicar com a aplicação através do uso da função de *callback*.

A arquitetura geral da aplicação pode ser vista na figura 4.3.

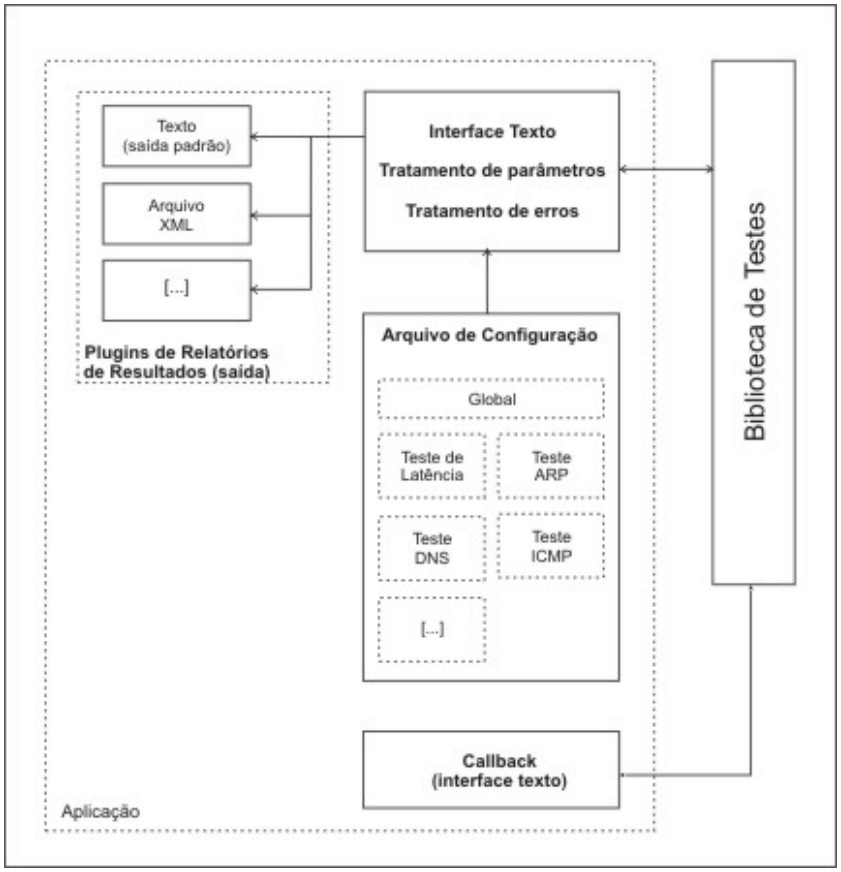

Figura 4.3: *Arquitetura Geral da Aplicação*

Nesta figura, percebe-se a segmentação dos procedimentos em três fases. Na primeira fase ocorre a carga dos dados do arquivo de configuração, o qual contém os parâmetros utilizados na execução de cada um dos testes. Na segunda fase ocorre a ativação dos testes pedidos usando os parâmetros lidos e a execução dos mesmos prossegue até seu término ou cancelamento através da callback. Na última fase, os resultados obtidos são passados aos plugins de saída, cujo funcionamento será melhor explicado na seção correspondente.

4.6.1 Interface texto

A aplicação têm uma interface texto simples mas poderosa. Através de parâmetros passados na linha de comando, é possível escolher as funcionalidades presentes. Abaixo temos a tela de ajuda da aplicação, onde as opções existentes são listadas:

```
sniffdet 0.7
A Remote sniffer Detection Tool
Copyright (c) 2002
   Ademar de Souza Reis Jr. <myself@ademar.org>
   Milton Soares Filho <eu_mil@yahoo.com>

Usage: ./sniffdet [options] TARGET
  Where:
  TARGET is a canonical hostname or a dotted decimal IPv4 address

  -i  --iface=DEVICE     Use network DEVICE interface for tests
  -c  --configfile=FILE  Use FILE as configuration file
  -l  --log=FILE         Use FILE for tests log
  -f  --targetsfile=FILE Use FILE for tests target
      --pluginsdir=DIR   Search for plugins in DIR
  -p  --plugin=FILE      Use FILE plugin
  -u  --uid=UID          Run program with UID (after dropping root)
  -g  --gid=GID          Run program with GID (after dropping root)

  -t  --test=[testname]  Perform specific test
      Where [testname] is a list composed by:
        dns         DNS test
        arp         ARP response test
        icmp        ICMP ping response test
        latency     ICMP ping latency test

  -v  --verbose          Run in verbose mode
  -h, --help             Show this help screen and exit
      --version          Show version info and exit

Defaults:
    Interface: "eth0"
    Log file: "sniffdet.log"
    Config file: "/etc/sniffdet.conf"
    Plugins Directory: "/usr/lib/sniffdet/plugins"
    Plugin: "stdout.so"

You have to inform at least one test to perform
```

O apêndice A traz a página manual relativa à utilização da aplicação *sniffdet*.

4.6.2 Plugins de Relatórios de Resultados

Cada administrador tem uma predileção quanto à maneira de se armazenar ou visualizar os resultados obtidos pelos testes de detecção. Bases de dados, registros de sistema, arquivos XML ou simplesmente arquivos em formato texto são alguns exemplos de locais onde tais resultados podem ser guardados. Para facilitar o atendimento a todos estes diferentes gostos, a aplicação dá suporte ao conceito de plugins, que são objetos executáveis que contém uma interface padronizada de ativação. Cada plugin oferecido pode ser relacionado a um destes diferentes métodos de armazenamento, sendo que novos podem ser adicionados sem que haja a necessidade de recompilação de qualquer parte do sistema. Atualmente, um plugin de visualização em modo texto e outro com saída em XML são disponibilizados.

Um exemplo de visualização de resultado através de um plugin em modo texto pode ser visto abaixo.

```
------------------------------------------------------------
Sniffdet Report
Generated on: Mon Oct 14 19:54:35 2002
------------------------------------------------------------
Tests Results for target 192.168.1.1
------------------------------------------------------------
Test: ARP Test
      Check if target replies a bogus ARP request (with wrong MAC)
Validation: OK
Started on: Mon Oct 14 19:54:34 2002
Finished on: Mon Oct 14 19:54:35 2002
Bytes Sent: 84
Bytes Received: 60
Packets Sent: 2
Packets Received: 1
------------------------------------------------------------
RESULT: POSITIVE
------------------------------------------------------------

------------------------------------------------------------
Number of tests with positive result: #1
------------------------------------------------------------
```

O conteúdo do arquivo gerado pelo plugin de saída em XML pode ser visto abaixo.

```
<?xml version="1.0"?>
<SNIFFDET-SESSION>
<info>
    <name>Latency test</name>
    <description>Ping response with custom packet flood</description>
    <validation>VALID</validation>
    <start-time>Wed Oct 30 01:16:37 2002</start-time>
```

```
    <finish-time>Wed Oct 30 01:17:02 2002</finish-time>
    <bytes-sent>337629680</bytes-sent>
    <bytes-received>0</bytes-received>
    <pkts-sent>3246439</pkts-sent>
    <pkts-received>13</pkts-received>
    <results unit="msecs">
        <normal>0.6</normal>
        <minimal>9.1</minimal>
        <maximal>548.9</maximal>
        <mean>168.4</mean>
    </results>
</info>
</SNIFFDET-SESSION>
```

4.6.3 Arquivo de Configuração

Uma das características mais importantes da biblioteca é sua flexibilidade. Pensando em aproveitar este poder e facilitar a utilização do sistema, a aplicação permite sua configuração através da utilização de um arquivo externo, no qual todos os parâmetros utilizados podem ser facilmente editados e adaptados de acordo com a necessidade do usuário. Além de toda a flexibilidade do arquivo de configuração, é possível ativar diversas opções através de parâmetros passados através da linha de comando.

Um exemplo de arquivo de configuração, assim como sua documentação estão disponíveis no apêndice A.

4.6.4 Segurança

Quanto à segurança, é importante frisar que a aplicação, apesar de poder ser executada somente por um usuário com privilégios de administrador, abdica de tais privilégios após a inicialização da interface, diminuindo o impacto da exploração de potenciais vulnerabilidades existentes no código.

4.7 Considerações Pós-desenvolvimento

Foi grande o desafio de se implementar um projeto que, além de funcionar em um ambiente de rede de computadores, toca em tantos subsistemas do núcleo do sistema operacional (rede, sistema de arquivos, *timers*, *threads* e outros). É claro que tal desafio seria muito mais complicado se não houvessem tantas ferramentas livres para auxiliar no desenvolvimento, em especial, ferramentas de diagnóstico de rede e de depuração de código.

Apesar da dificuldade, o produto final consegue agrupar várias qualidades necessárias para, ao menos, interessar pela sua utilização e estudo. Dentre o público que pode ser tocado pela existência deste software, pode-se destacar os administradores de rede, profissionais da área de segurança e estudantes cursando disciplinas intermediárias sobre redes de computadores.

Até o momento, o *sniffdet* é a única ferramenta que agrupa as seguintes qualidades:

Licença GPL Este tipo de licença encoraja o estudo e utilização por tornar o sistema gratuito e permitir modificações em código. Outro fator importante é que este tipo de software tende a evoluir rapidamente, pois conta com a crítica e sugestão constante de seu público usuário.

Flexibilidade A preocupação com a descaracterização de assinaturas, liberdade de configuração e poder de ajuste aos usuários através do uso de arquivos de configuração e parâmetros passados pela linha de comando garantem uma vida útil maior ao sistema. Também neste tópico se destaca a possibilidade de inserção de novos plugins de relatórios de resultados, tornando o sistema ainda mais adaptado à necessidade de seu usuário.

Segurança Além de ser mantida em mente a todo instante ainda pode ser testada através da auditoria do código, por ser aberto.

Simplicidade na Utilização A interface em modo texto foi desenvolvida de maneira a ser simples e intuitiva.

Um fator preocupante para a implementação é a possível heterogeneidade de sistemas e plataformas que trabalham numa rede. Estas diferentes configurações, tanto de software quanto de hardware, dificultam a generalização e adaptação dos parâmetros dos testes de detecção. Somente através de sucessivos testes utilizando várias configurações diferentes é possível chegar a um ajuste próximo do ideal. Por causa disso, a preparação do próximo capítulo levou nossa atenção de volta ao estágio de implementação, tornando-se parte do processo evolutivo do sistema.

Capítulo 5

Experimentos e Resultados

Como mostrado no capítulo anterior, foi desenvolvida uma biblioteca de testes e uma aplicação para a detecção remota de *sniffers*. Neste capítulo relatamos os resultados de tais testes, assim como suas limitações, confiabilidade e fatores que podem influenciar novos experimentos.

5.1 Ambientação

Os experimentos aqui relatados foram realizados em ambiente doméstico. A variedade de equipamentos e softwares utilizados é pequena, porém dado que a aplicação é de livre distribuição, espera-se que esta seja amplamente testada e aperfeiçoada por usuários e interessados após o seu lançamento público.

Entre os fatores que mais podem influenciar nos resultados dos testes, podemos citar o sistema operacional utilizado e as configurações deste, o *driver* da interface e a capacidade da rede, assim como todos os equipamentos e software envolvidos na captura e geração de tráfego na rede.

Os equipamentos empregados nos experimentos estão listados abaixo:

- Microcomputador AMD Duron 1GHz, 256MB RAM
- Microcomputador Intel Pentium 200MHz, 64MB RAM
- Interfaces de rede PCI 100Mbps, chipset RTL8139
- Interfaces de rede PCI 10Mbps, chipset RTL8129
- Modem ADSL US Robotics USR8550
- Switch Encore ENH908-NWY+
- HUB 10Mbps genérico

- Cabos de rede coaxiais e cabos padrão RJ45

Em termos de software, foram utilizados três sistemas operacionais para os testes: Linux (kernel 2.2 e 2.4), FreeBSD 4.2 e Microsoft Windows 98SE. Dado o pouco domínio sobre estes dois últimos sistemas por parte dos autores, os resultados neles obtidos são menos conclusivos que os obtidos no sistema Linux, extensivamente utilizado para o desenvolvimento do projeto.

A lista de softwares utilizados nos experimentos encontra-se abaixo:

- Sistemas Operacionais: GNU/Linux (kernels 2.4.18 e 2.2.19), FreeBSD 4.2 e Microsoft Windows 98se.
- *sniffers*: Ethereal 0.9.6 e tcpdump 3.7.1

Os *sniffers* utilizados são bastante flexíveis, oferecendo opções para filtragem de pacotes, visualização de conteúdo em tempo real, resolução reversa de nomes, várias opções para gravação dos dados, etc. Sendo assim, foi possível reproduzir o comportamento de sniffers reais a partir dessas opções de configuração sem grandes dificuldades.

5.2 Descrição dos Resultados

Devido a característica de flexibilidade do projeto, permitindo a alteração dos parâmetros utilizados sem a necessidade de recompilação, conseguiu-se uma quantidade significativa de informações na realização dos testes, porém, demasiadamente grande para serem descritas por completo neste documento.

Nas seções a seguir apresentamos exemplos de execução e particularidades relevantes a cada um dos testes implementados.

5.2.1 Teste ICMP

O teste de requisições ICMP com endereço de hardware falso é bastante simples. Como o objetivo é descobrir se determinada interface de rede da máquina alvo está operando em modo promíscuo, o *sniffer* utilizado não é relevante. Na verdade, qualquer ferramenta que configure a interface para trabalhar em modo promíscuo pode ser utilizada.

Em relação aos parâmetros do teste, foram utilizados os seguintes valores:

- Tempo limite de execução: 20 segundos.

- Número máximo de requisições enviadas: 10.
- Intervalo entre requisições: 1 segundo.
- Falso endereço de hardware (Ethernet MAC): `{0xff,0x00,0x00,0x00,0x00,0x00}`.
- Pacote ICMP: Requisição ICMP simples, idêntica à utilizada pelo utilitário *ping*, disponível em diversos sistemas operacionais.

Todos esses valores são configuráveis pela aplicação de testes, e podem ser alterados conforme as características das máquinas e da rede envolvidas.

A utilização de valores de endereço de hardware iniciados em `0xff` se mostrou necessária para que este teste funcionasse, conforme justificativa dada em 3.3.1.

Em todas as execuções desse teste, resultados positivos foram retornados logo após o envio dos primeiros pacotes, frequentemente nos primeiros 2 segundos após a ativação do teste de detecção. A carga da rede não tem influência nos testes. Testes em máquinas com interfaces em modo normal não reportaram falso positivo, retornando sempre após o tempo máximo configurado para o teste.

É importante ressaltar que esse teste não se mostrou efetivo na detecção de interfaces em modo promíscuo no ambiente Windows por nós testado. Acredita-se que o *driver* da interface em questão tenha algum tipo de filtro embutido, não repassando os pacotes para as camadas superiores da pilha TCP/IP.

Abaixo temos uma pequena lista de execuções:

- Linux 2.4: Execução e detecção ocorreram sem problemas, utilizando qualquer endereço de hardware como destino.
- FreeBSD 4.2: Execução e detecção ocorreram sem problemas, utilizando qualquer endereço de hardware como destino.
- Microsoft Windows 98: Não houve detecção, resultados não conclusivos.

5.2.2 Teste ARP

O teste de requisições ARP com endereço de hardware falso também é bastante simples e assim como o teste ICMP, apenas detecta se a interface da máquina alvo está em modo promíscuo. Foram utilizados os seguintes valores para o teste de requisição ARP:

- Tempo limite de execução: 20 segundos.

- Número máximo de requisições enviadas: 10.
- Intervalo entre requisições: 1 segundo.
- Falso endereço de hardware (Ethernet MAC): `{0xff,0x00,0x00,0x00,0x00,0x00}`.
- Requisição ARP comum, idêntica às geradas pelo sistema operacional Linux 2.4 na rede observada.

Todos esses valores são configuráveis pela aplicação de testes, e podem ser alterados para se adequar a particularidades da rede e do ambiente de execução.

Assim como no teste ICMP, foi necessária a utilização de um endereço de hardware iniciado em `0xff` para que esse teste se mostrasse efetivo.

Em todas as execuções do teste, resultados positivos eram retornados logo após o envio das primeiras requisições, também frequentemente nos primeiros 2 segundos após a ativação do teste de detecção. Esse teste se mostrou efetivo na detecção de interfaces em modo promíscuo nos três sistemas operacionais testados, porém falsos positivos podem ser reportados caso haja tráfego de rede legítimo entre as duas máquinas, uma vez que não é possível diferenciar uma resposta ARP legítima de uma gerada pela requisição falsa.

Abaixo temos uma pequena lista de execuções:

- Linux 2.4: Execução e detecção ocorreram sem problemas.
- FreeBSD 4.2: Execução e detecção ocorreram sem problemas.
- Microsoft Windows 98: Detecção sem problemas.

5.2.3 Teste DNS

Para que esse teste seja efetivo, o *sniffer* deve ter a resolução reversa de nomes ativada. Essa opção é ligada por padrão no Ethereal e em algumas versões do tcpdump. Em ambos a opção pode ser habilitada a partir da linha de comando ou da interface gráfica.

Os pacotes utilizados nas conexões falsas são do protocolo TCP e tem vários dos seus campos de opção preenchidos com valores configuráveis. Em nossos experimentos utilizamos as seguintes opções:

- Tempo limite de execução: 20 segundos.
- Número de pacotes enviados: 10.
- Intervalo entre o envio: 1 segundo.

- Falso endereço de hardware (Ethernet MAC): arbitrário.
- Falso endereço IP: `10.0.0.21`
- Porta de conexão (destino): `23(telnet)`
- Porta de conexão (origem): `23`
- Porção de dados do pacote *(payload)*: VAZIO

Todos os valores foram escolhidos de maneira arbitrária. A alteração destes se mostra útil em casos particulares, como quando da necessidade de evitar a caracterização do teste ou adequá-lo a um determinado ambiente, mas não é necessária na maioria dos casos.

Comportamento dos Sistemas Operacionais testados:

- Linux 2.4: Execução e detecção ocorreram sem problemas.
- FreeBSD 4.2: Execução e detecção ocorreram sem problemas.
- Microsoft Windows 98: Detecção sem problemas.

5.2.4 Teste de Latência

Como o teste de latência não é determinístico, a análise dos resultados deve levar em consideração várias características do ambiente testado. Uma vez que o objetivo do teste é estressar a máquina alvo, um conhecimento da arquitetura e funcionamento interno do sistema operacional e do *sniffer* destino se mostra extremamente útil na escolha dos valores utilizados nesse teste e a quem está a interpretar os resultados.

Encontrar a combinação ótima de parâmetros e tráfego a ser utilizado por esse teste requer um estudo que foge do escopo deste trabalho. Com a ferramenta disponibilizada, novos experimentos podem ser realizados e aperfeiçoados num futuro próximo.

Em nossos experimentos, utilizamo-nos de uma sequência de pacotes TCP com a flag SYN ativada e com a porção de dados vazia ou de tamanho bastante reduzido[1]. O objetivo da utilização de pacotes com essas característica foi forçar o *sniffer* a processar um grande número de cabeçalhos de pacotes. Essa abordagem funcionou bem no sistema operacional Linux, mas não gerou resultados conclusivos nos outros dois sistemas testados (MS Windows 98 e FreeBSD 4.2).

Os valores para o endereço de hardware (MAC) e endereço IP foram escolhidos arbitrariamente, e, se necessário, podem ser alterados para se adequar às características da rede e dos *sniffers* testados.

[1]Não faz sentido um pacote que tenha a flag SYN ativada carregar dados, mas durante os testes não estamos necessariamente preocupados em gerar tráfego válido.

Teste 1

Máquina executando o teste:

- Processador AMD Duron 1GHz, 256MB RAM
- Interface de rede 100Mbps
- Sistema Operacional: Linux (kernel 2.4.18)

Máquina alvo do teste:

- Intel Pentium 200Mhz, 64MB RAM
- Interface de rede 100Mbps
- Sistema Operacional: Linux (kernel 2.4.18)
- *sniffer* em execução: tcpdump 3.7.1 (opções padrão)

Parâmetros utilizados:

- Tempo de execução do teste: 180 segundos
- Intervalo entre as requisições ICMP: 1 segundo

Pacote utilizado para a inundação:

- Falso endereço de hardware de origem: `{0x00,0x55,0x34,0x21,0x11,0xff}`
- Falso endereço de hardware de destino: `{0x00,0x44,0x34,0x2A,0x0B,0xff}`
- Falso endereço IP de origem: `10.0.0.21`
- Falso endereço IP de destino: `10.0.0.30`
- Porta de conexão (destino): `23(telnet)`
- Porta de conexão (origem): `23`
- Porção de dados do pacote (payload): 2 bytes, conteúdo arbitrário
- Pico do tráfego alcançado na rede: 20Mbps
- Responsividade da máquina executora do teste: aceitável; processamento normal
- Responsividade da máquina alvo do teste: muito baixa; processamento lento; dificuldade até mesmo para trocar de um terminal (tty) para outro

Os resultados obtidos com tal teste podem ser vistos abaixo:

Interface alvo em modo normal

- Requisições enviadas: 122
- Requisições respondidas: 122 **(100%)**
- Tempo de resposta normal: 0.2ms
- Tempo de resposta durante inundação (min/med/max): 0.2/0.2/1.3 (ms)

Interface alvo em modo promíscuo

- Requisições enviadas: 122
- Requisições respondidas: 44 **(36%)**
- Tempo de resposta normal: 0.2ms
- Tempo de resposta durante inundação (min/med/max): 1.0/1.5/6.3 (ms)

Como pode ser visto acima, a máquina alvo quando com um *sniffer* em execução em modo promíscuo não foi capaz de responder todas as requisições ICMP (obtivemos variações entre 30 e 50% de responsividade), e quando o fez, foi com uma demora considerável em relação ao mesmo teste com a interface em modo normal. Esse teste foi repetido várias vezes e os resultados levaram às mesmas conclusões, mesmo sob condições ligeiramente diferentes (tráfego na rede, utilização da máquina para outros fins, etc).

5.2.5 Comportamentos Inesperados

Como nossa implementação insere uma grande quantidade de pacotes “estranhos” no barramento da rede, alguns equipamentos reagiram de maneira inesperada à execução de alguns testes:

Modem ADSL: A cada execução do teste de latência, o modem ADSL, que estava no mesmo barramento da rede, perdia a conexão externa com a Internet. O fabricante foi contactado a respeito desse comportamento, mas até o presente momento não obtivemos resposta. Além disso, a interface do modem conectada à rede interna parece estar em modo promíscuo, uma vez que responde positivamente aos testes ICMP e ARP.

Switch: O switch utilizado é bastante simples e apresenta pouca robustez. Durante a inundação de pacotes do teste de latência, este diversas vezes passou a se comportar como um *HUB*, retransmitindo todos os pacotes para todas as máquinas a ele conectadas.

Máquinas pertencentes ao barramento da rede testado não apresentaram comportamento inesperado durante a execução dos testes ICMP, DNS e ARP. Já na execução do teste de latência, máquinas com interfaces em modo promíscuo tiveram sua responsividade diminuida, uma vez que nesses casos o número de pacotes processados pela pilha de rede do sistema operacional era grande.

Capítulo 6

Conclusão

sniffers são geralmente executados a partir de máquinas comprometidas por atacantes e injetam o mínimo de pacotes na rede em que estão sendo executados. Esse *modus operandi* faz com que sistemas de detecção locais percam sua confiabilidade, pois o atacante pode facilmente camuflar a existência de seu *sniffer* ao manipular os sistemas que indicam sua utilização, inclusive modificando o funcionamento de módulos internos do núcleo do sistema operacional. Já sistemas de detecção remota devem levar isto em consideração e esforçar-se ao máximo em termos de flexibilidade para que *sniffers* mais avançados não venham a reconhecer um padrão na sua utilização e desenvolver uma técnica de evasão.

Apesar da criptografia e da limitação de tráfego imposta pelo cuidado na escolha da configuração da rede restringir bastante a eficência de um *sniffer*, ainda há muita informação importante que pode ser angariada, o que não permite que somente tais técnicas sejam adotadas em substituição à utilização de sistemas de detecção remota de *sniffers*.

Tanto a bibloteca como a aplicação implementadas são abertas e têm seu código fonte disponível na Internet. Aliadas à essa característica estão flexibilidade, simplicidade e segurança, o que fazem de nossa implementação um bom exemplo para ser utilizado por administradores de rede e profissionais da área de segurança, além de servir como interessante ferramenta de estudo para alunos de cursos de redes de computadores.

A implementação de um sistema de detecção remota de *sniffers* é complexa pois sua eficácia é sensível a fatores externos, tais como o funcionamento das implementações dos protocolos da pilha TCP/IP e dos equipamentos usados na rede que podem ser muitos variados e diferenciados. A indisponibilidade de um laboratório para testes fez com que, infelizmente, os resultados obtidos com a flexibilidade da aplicação tenham sido mais téoricos do que práticos.

A segurança de um sistema depende de um conjunto considerável de fatores. Não existe solução ótima que aborde todos os pontos vulneráveis, assim como não existe detector perfeito que possa detectar todo e qualquer tipo de *sniffer*. O cenário parece-se com o de uma competição

entre os atacantes e responsáveis por segurança, e o objetivo destes últimos é sempre estar à frente, dificultando o máximo possível o caminho dos que pretendem comprometer a estrutura ou apoderar-se de dados importantes.

Trabalhos Futuros

Ainda há muito o que ser estudado e desenvolvido na área de detecção de *sniffers*. Como toda área em evolução, novos ambientes, topologias e arquiteturas de rede estão sempre surgindo e os atacantes frequentemente utilizando sua infinita criatividade para contornar as barreiras a eles impostas.

Tanto a aplicação como a biblioteca desenvolvida ainda estão em estágio inicial e, além de muitos testes e ajustes finos, têm várias funcionalidades a serem adicionadas. Aumentar o número de testes disponíveis, assim como incluir diversas variações dos existentes e fazer estudos qualitativos sobre seu funcionamento em diversas plataformas seriam os principais focos de extensão deste trabalho. Por parte da aplicação, é interessante desenvolver novos *plugins* de saída, assim como desenvolver uma interface gráfica e sistemas de reação e resposta aos resultados obtidos.

O lançamento público do código fonte de toda a implementação, assim como a extensa documentação disponível, devem permitir que o desenvolvimento e testes do projeto sejam acelerados num futuro próximo.

Apêndice A

Páginas de Manual *(”Unix Manpages”)*

Este apêndice contém as páginas de manual do programa *sniffdet* e da biblioteca *libsniffdet* versão 0.7. Como o software tem um alcance mundial (é de livre distribuição), seu desenvolvimento e documentação foram feitos primariamente em inglês, e esse é o idioma na qual as páginas de manual a seguir estão escritas.

A.1 libsniffdet (3)

```
LIBSNIFFDET(3)     Remote Sniffer Detection Library     LIBSNIFFDET(3)

NAME
   libsniffdet - Sniffer detection library

DESCRIPTION
   This library is useful for remote sniffer detection or to discover
   machines which are running in promiscuous mode. You can see the
   full documentation at http://sniffdet.sourceforge.net

SYNOPSIS
   #include <sniffdet.h>

GENERAL DEFINITIONS

   CALLBACK

   The callback functions used by the detection tests for activity
   report and interactivity issues must have the following prototype,
   providing that its return value is used to cancel the current
   execution of the detection test.

   int (*user_callback)(struct test_status *status, int msg_type, char
   *msg);

   The first argument is a structure of the type below, containing
   information about the state of execution (in percent) and the
```

```
quantity of incoming and outcoming packets of the current test.

struct test_status {
       unsigned short int percent; // 0% to 100%
       unsigned int bytes_sent;
       unsigned int bytes_recvd;
};

The second argument is one of the following enumerations.

RUNNING - used just for resposivity purposes
NOTIFICATION - general messages
ERROR - critical conditions (abort cases)
WARNING - critical conditions (do not abort the execution)
DETECTION - detection performed
ENDING - indicates the end of the detction test

DEVICE

The following functions should be used to initialize/finish the
network device.

struct sndet_device * sndet_init_device(
       char *device,
       int promisc,
       char *errbuf);

int sndet_finish_device(
       struct sndet_device *device,
       char *errbuf);

Where struct sndet_device has the following layout:

struct sndet_device {
       char *device;
       int datalink;
       int pkt_offset;
       struct libnet_link_int *ln_int;
       pcap_t *pktdesc;
       bpf_u_int32 network;
       bpf_u_int32 netmask;
       int rawsock;
};

// datalink type
// device name
// sync bytes
// raw socket id

RESULTS

All the detection tests return their results in the following
structure.

struct test_info {
       enum test_code code;
       int valid;
       char *test_name;
       char *test_short_desc;
```

```
        time_t time_start;
        time_t time_fini;
        unsigned int b_sent;
        unsigned int b_recvd;
        unsigned int pkts_sent;
        unsigned int pkts_recvd;
        union {
                struct icmptest_result icmp;
                struct arptest_result arp;
                struct dnstest_result dns;
                struct latencytest_result latency;
        } test;
};

// detection test enumeration - see libsniffdet.h
// wether it was valid or not
// name of the test
// test short description
// start time
// stop time
// bytes sent
// bytes received
// packets sent
// packets received
// specifics results

GENERAL USE FUNCTIONS

There are many functions built to provide basic network and
general purpose functions.

u_long sndet_resolve(char *hostname);
    Resolve hostname, returns binary representation in
    network-ordered representation. Hostname is an ASCII string
    representing an IPv4 address (canonical hostname or doted
    decimal representation).

int sndet_random(void);
    Returns a pseudo random integer

int sndet_ping_host(
    Common ping function. Provided are the target name (host), a
    pointer to the interface structure (device), the timeout in
    seconds, the interval between target probes (send_interval)and
    the amount of packets sent on each probe (burst_size). The last
    two args are used to return the results and to write the error
    message in case an internal error occurs. It returns non-zero
    if any error occurs.

u_long sndet_get_iface_ip_addr(
    Returns interface IP address in binary notation (host-ordered)
    for the given interface structure (sndet).  If any error
    occurs, an error message will be writen in errbuf.

struct ether_addr * sndet_get_iface_mac_addr(
    Returns interface MAC address

unsigned char *sndet_gen_tcp_pkt(
    Generates a TCP packet based on information supplied in
```

```
       custom_pkt information

   void sndet_sleep(long sec, long usec);
       Independent and portable way for sleeping

DETECTION TESTS
   The folowing are the detection test implemented by the library.
   They always have as obrigatory arguments the name of the target
   host and the device structure. The rest of theirs parameters will
   be replaced for internal values if not specified (passing NULL or
   zero, depending of the data type). As a general rule, all the tests
   return non-zero if an error occurs. For more specific information
   about the error, one should verify the message returned by the
   callback functions.

   ICMP TEST
   int sndet_icmptest(
          char *host,
          struct sndet_device *device,
          unsigned int tmout,
          unsigned int tries,
          unsigned int send_interval,
          user_callback callback,
          struct test_info *result,
          char *fakehwaddr
   );

   // suspicious host
   // timeout in seconds
   // max number of tries
   // interval between packets sent (in msec)
   // fake MAC hardware address sent to the host

   ARP TEST
   int sndet_arptest(
          char *host,
          struct sndet_device *device,
          unsigned int tmout,
          unsigned int tries,
          unsigned int send_interval,
          user_callback callback,
          struct test_info *result,
          char *fakehwaddr
   );

   // suspicious host
   // timeout in seconds
   // max number of tries
   // interval between packets sent (in msec)
   // fake MAC hardware address sent to the host

   DNS TEST
   int sndet_dnstest(
          char *host,
          struct sndet_device *device,
          unsigned int tmout,
          unsigned int tries,
          unsigned int send_interval,
          user_callback callback,
```

```
           struct test_info *info,
           // bogus pkt information, optional
           char *fake_ipaddr,
           char *fake_hwaddr,
           ushort dport, ushort sport,
           char *payload,
           short int payload_len
   );

   // pkt source
   // pkt destination
   // destination/source port
   // payload data
   // payload length

   LATENCY TEST
   int sndet_latencytest_pktflood(
          char *host,
          struct sndet_device *device,
          unsigned int tmout,
          unsigned int probe_interval,
          user_callback callback,
          struct test_info *info,
          struct custom_info *bogus_pkt
   );

   // suspicious host
   // timeout in seconds
   // interval between probes (x10 msec)
   // info about the fake packet desired

   As the result, there's the structure below (time measured as
   tenths of second and RTT = Round Trip Time).

   struct latencytest_result {
          // time is expressed in msec/10
          u_int normal_time;
          u_int min_time;
          u_int max_time;
          u_int mean_time;
   };


EXAMPLES
   See the documentation included with the library and the source
   distribution, which you can found at
   http://sniffdet.sourceforge.net

BUGS
   This library is in beta stage and is not widely tested. Your
   support is appreciated. :-)

   Please report bugs at http://sniffdet.sourceforge.net or to
   sniffdet-devel@lists.sourceforge.net

   Also take a look in our TODO file.

COPYRIGHT
   Copyright (c) 2002
```

```
      Ademar de Souza Reis Jr. <myself@ademar.org>
      Milton Soares Filho <eu_mil@yahoo.com>

SEE ALSO
   sniffdet(1) libnet(3) pcap(3)
   http://sniffdet.sourceforge.net

sniffdet manpage               2002-11-28              LIBSNIFFDET(3)
```

A.2 sniffdet (1)

```
SNIFFDET(1)          Remote Sniffer Detection Tool         SNIFFDET(1)

NAME
   sniffdet 0.7 - Remote sniffer detection tool

SYNOPSIS
   sniffdet [options] TARGET

DESCRIPTION
   Sniffdet is an OpenSource implementation of a set of tests for
   remote sniffers detection in TCP/IP network environments. It is
   useful for remote sniffer detection or to just discover machines
   which are running in promiscuous mode.

   Sniffdet is very flexible and allows you to configure many of its
   options by using the config file /etc/sniffdet.conf. It also has
   plugins support for the result of its tests (currently, XML and
   stdout output are created).

   You can see the full documentation at
   http://sniffdet.sourceforge.net

OPTIONS
   TARGET is a canonical hostname or a dotted decimal IPv4 address

   -i --iface=DEVICE
        Use network DEVICE interface for tests.
        Default is eth0 in linux systems.

   -l --log=FILE
        Use FILE for tests log.
        Default is none

   -c --configfile=FILE
        Use FILE as configuration file for application.
        Default is /etc/sniffdet.conf

   -f --hostsfile=FILE
        Use FILE as input for tests target. The file must be in
        ascii with one hostname, IP or net address per line.
        Comments start with '#'

   -u --uid=UID
        Run program with UID (after dropping root).
        Default is UID 280 (from config file)

   -g --gid=GID
        Run program with GID (after dropping root)
        Default is GID 280 (from config file)

   -t --test=[testname]
        Perform a specific test(s)
        Where [testname] is a list composed by at least one of:
            dns         DNS test
            arp         ARP response test
```

```
            icmp        ICMP ping response test
            latency     ICMP ping latency test

         See the full documentation included with the
         library for information about all tests

   --pluginsdir=[directory]
         Select a directory where sniffdet will load plugins from

   -p --plugin=[plugin_name]
         Select a plugin to load (xml, stdout, etc).

   -f --targetsfile=[file]
         Scan all targets present in a file with a test.

   -v --verbose
         Run in verbose mode (extra output messages).
         Default is no.

   -s --silent
         Run in silent mode (no output messages).
         Default is no.

   -h --help
         Show a help screen and exit

   --version
         Show version information and exit

EXAMPLES
   # sniffdet -i eth1 -t dns,arp,icmp foo.localdomain

   Test the host foo.localdomain with dns, arp and icmp tests using
   the interface eth1

   # sniffdet -i eth0 -t latency foo.localdomain --plugin=xml

   Test the machine foo.localdomain using the latency test through the
   interface eth0. Output results using the xml plugin.

BUGS
   This program is in beta stage and is not widely tested. Your
   support is appreciated. :-)

   Please report bugs at http://sniffdet.sourceforge.net or to
   sniffdet-devel@lists.sourceforge.net

   Also see our TODO file.

COPYRIGHT
   Copyright (c) 2002
      Ademar de Souza Reis Jr. <myself@ademar.org>
      Milton Soares Filho <eu_mil@yahoo.com>

SEE ALSO
   sniffdet.conf(5) libsniffdet(3)
   http://sniffdet.sourceforge.net

sniffdet manpage            2002-11-25                  SNIFFDET(1)
```

A.3 sniffdet.conf (2)

```
SNIFFDET(1)          Remote Sniffer Detection Tool         SNIFFDET(1)

NAME
   sniffdet.conf - sniffdet configuration file

DESCRIPTION
   sniffdet.conf allows you to configure the way sniffdet performs its
   tests. It’s located in /etc by default and has various sections,
   all described below.

SYNTAX
   The syntax is very simple. Each section has a name and is delimited
   by brackets "{}". Inside the section, simple attributions are made
   to variables.

   Comments are started with "#" and can be located anywhere in the
   file. Everything after a "#" is ignored by the parser until a line
   break.

   Blank lines are ignored.

EXAMPLE
   An example of a configuration file follows (it’s  filled  with
   some default values from the current implementation of libsniffdet,
   but should not be used in production enviroments. We strongly
   recommend that you create your own config file to avoid
   identification of the tests by the sniffers.

   # snifdet example configuration file
   # http://sniffdet.sourceforge.net
   #
   # see sniffdet.conf (5) manpage

   # global configuration
   global {
        verbose = 0;
        # this is one or a combination of FILE, STDOUT, STDERR, SYSLOG
        logtype = FILE;
        # want a log by default?
        logfile = "sniffdet.log";
        #plugins_dir = "/usr/lib/sniffdet/plugins";
        plugin = "stdout.so";
        # UID to use after dropping root privileges
        UID = 280;
        # GID to use after dropping root privileges
        GID = 280;
        iface = "eth0";
        fake_hwaddr = {0xff, 0x00, 0x00, 0x00, 0x00, 0x00};
        fake_ipaddr = "192.168.1.100";
   }

   # icmp test variables
   icmptest {
        # interface per test not supported yet
        #iface = "eth0";
```

```
        timeout = 20; # secs
        tries = 10;
        interval = 1000 # msecs
        fake_hwaddr = {0xff, 0x00, 0x00, 0x00, 0x00, 0x00};
   }

   # arp test variables
   arptest {
        # interface per test not supported yet
        #iface = "eth0";
        timeout = 20; # secs
        tries = 10;
        interval = 1000 # msecs
        fake_hwaddr = {0xff, 0x00, 0x00, 0x00, 0x00, 0x00};
   }

   # dns test variables
   dnstest {
        # interface per test not supported yet
        #iface = "eth0";
        timeout = 20; # secs
        tries = 10;
        interval = 1000 # msecs
        fake_ipaddr = "10.0.0.10"
        fake_hwaddr = {0x46, 0x0f, 0xA4, 0x33, 0x11, 0xD1};
        sport = 22;
        dport = 22;
        # payload support not implemented in parser yet...
        #payload = "login: foobar";
   }

   # latency test variables
   latencytest {
        # interface per test not supported yet
        #iface = "eth0";
        timeout = 300; # secs
        interval = 1500; # msecs
        # tcpflags support not implemented in parser yet...
        #tcpflags = SYN;
        # payload support not implemented in parser yet...
        #payload = "";
   }
   # EOF

COPYRIGHT
   Copyright (c) 2002
      Ademar de Souza Reis Jr. <myself@ademar.org>
      Milton Soares Filho <eu_mil@yahoo.com>

BUGS
   - payload and tcpflags parser not implemented yet
   - multi-line support not implemented

SEE ALSO
   sniffdet(1) libsniffdet(3)
   http://sniffdet.sourceforge.net

sniffdet manpage            2002-11-28                  SNIFFDET(1)
```

Referências Bibliográficas

[1] Anderson e Needham. Robustness Principles for Public Key Protocols. *CRYPTO: Proceedings of Crypto*, 1995. Disponível em `http://citeseer.nj.nec.com/anderson95robustness.html`.

[2] Apostolopoulos, Peris, e Saha. Transport Layer Security: How Much Does it Really Cost? *INFOCOM: The Conference on Computer Communications, joint conference of the IEEE Computer and Communications Societies*, 1999. Disponível em `http://citeseer.nj.nec.com/apostolopoulos99transport.html`.

[3] Daniel J. Barrett e Richard E. Silverman. *SSH: The Secure Shell: The Definitive Guide*. O'Reilly & Associates, Inc., 2001.

[4] Steven M. Bellovin e Michael Merritt. Limitations of the Kerberos Authentication System. *USENIX Conference Proceedings*, páginas 253–267, Dallas, TX, 1991. USENIX. Disponível em `http://citeseer.nj.nec.com/article/bellovin91limitations.html`.

[5] Peter Breuer. *Linux Bridge+Firewall Mini-HOWTO*, dezembro de 1997.

[6] Computer Emergency Response Team CERT. Trends in Denial of Service Attack Technology, outubro de 2001. Disponível em `http://www.cert.org/archive/pdf/DoS_trends.pdf`.

[7] Computer Emergency Response Team CERT. Social Engineering Attacks via IRC and Instant Messaging, março de 2002. Disponível em `http://www.cert.org/incident_notes/IN-2002-03.html`.

[8] Netscape Comunications. The SSL Protocol Version 3.0, novembro de 1996. Disponível em `http://wp.netscape.com/eng/ssl3/draft302.txt`.

[9] M. Crispin. Internet Message Access Protocol - Version 4rev1. ARPA RFC - 2060, dezembro de 1996. Disponível em `ftp://ftp.rfc-editor.org/in-notes/rfc2060.txt`.

[10] CVS Project. Concurrent version system - cvs home, 2002. Disponível em `http://www.cvshome.org`.

[11] Steve Deering. Host Extensions for IP Multicasting. ARPA RFC - 1112, agosto de 1989. Disponível em `ftp://ftp.rfc-editor.org/in-notes/rfc1112.txt`.

[12] T. Dierks e C. Allen. The TLS Protocol. ARPA RFC - 2246, janeiro de 1999. Disponível em `ftp://ftp.rfc-editor.org/in-notes/rfc2246.txt`.

[13] The Honeynet Project (Editor). *Know Your Enemy: Revealing the Security Tools, Tactics, and Motives of the Blackhat Community*. Addison-Wesley Pub Co, 1st edition, 2001.

[14] J. Case et al. A Simple Network Management Protocol (SNMP). ARPA RFC - 1157, maio de 1990. Disponível em `ftp://ftp.rfc-editor.org/in-notes/rfc1157.txt`.

[15] R. Fielding et al. Hypertext Transfer Protocol – HTTP/1.1. ARPA RFC - 2616, junho de 1999. Disponível em `ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt`.

[16] Ethereal Project. Ethereal network analyser, version 0.9.7, 2002. Disponível em `http://www.ethereal.com`.

[17] Free Software Foundation. GNU General Public License Version 2, junho de 1991. Disponível em `http://www.gnu.org/copyleft/gpl.html`.

[18] Thomas Glaser. TCP/IP Stack Fingerprinting Principles, outubro de 2000. Disponível em `http://www.sans.org/newlook/resources/IDFAQ/TCP\_fingerprinting.htm`.

[19] B. Gleeson, A. Lin, J. Heinanen, e G. Armitage. A framework for IP based virtual private networks. ARPA RFC - 2764, agosto de 1998. Disponível em `http://citeseer.nj.nec.com/gleeson00framework.html`.

[20] Robert Graham. Sniffing (network wiretap, sniffer) FAQ, setembro de 2000. Disponível em `http://www.robertgraham.com/pubs/sniffing-faq.html`.

[21] J. Kohl e C. Neuman. The Kerberos Network Authentication Service (V5). ARPA RFC - 1510, setembro de 1995. Disponível em `ftp://ftp.rfc-editor.org/in-notes/rfc1510.txt`.

[22] Joe Loughry e David A. Umphress. Information Leakage from Optical Emanations. *ACM Transactions on Information and Systems Security*, 5(3), agosto de 2002. Disponível em `http://citeseer.nj.nec.com/528578.html`.

[23] Meadows. A Formal Framework and Evaluation Method for Network Denial of Service. *PCSFW: Proceedings of The 12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999. Disponível em `http://citeseer.nj.nec.com/meadows99formal.html`.

[24] J. Moore. Protocol Failures in Cryptosystems. *Proceedings of the IEEE*, 5(76):594–602, agosto de 1988.

[25] Pars Mutaf. Defending against a Denial-of-Service Attack on TCP. *Recent Advances in Intrusion Detection*, 1999. Disponível em `http://citeseer.nj.nec.com/mutaf99defending.html`.

[26] J. Myers e M. Rose. Post Office Protocol - Version 3. ARPA RFC - 1939, maio de 1996. Disponível em `ftp://ftp.rfc-editor.org/in-notes/rfc1939.txt`.

[27] J. Oikarinen e D. Reed. Internet Relay Chat Protocol. ARPA RFC - 1459, maio de 1993. Disponível em `ftp://ftp.rfc-editor.org/in-notes/rfc1459.txt`.

[28] David C. Plummer. An Ethernet Address Resolution Protocol. ARPA RFC - 826, novembro de 1982. Disponível em `ftp://ftp.rfc-editor.org/in-notes/rfc826.txt`.

[29] J. Postel e J. Reynolds. Telnet Protocol Specification. ARPA RFC - 854, maio de 1983. Disponível em `ftp://ftp.rfc-editor.org/in-notes/rfc854.txt`.

[30] J. Postel e J. Reynolds. File Transfer Protocol (FTP). ARPA RFC - 959, outubro de 1985. Disponível em `ftp://ftp.rfc-editor.org/in-notes/rfc959.txt`.

[31] Jonathan B. Postel. Simple Mail Transfer Protocol. ARPA RFC - 821, agosto de 1982. Disponível em `ftp://ftp.rfc-editor.org/in-notes/rfc821.txt`.

[32] Jonathan J. Rusch. The 'Social Engineering' of Internet Fraud. *9th Annual Conference of the Internet Society*, 1999. Disponível em `http://www.isoc.org/isoc/conferences/inet/99/proceedings/3g/3g_2.htm`.

[33] Modulo Security Solutions S.A. 7ª Pesquisa Nacional de Segurança da Informação, julho de 2001. Disponível em `http://www.modulo.com.br/pdf/pesq_seg_01.zip`.

[34] Stefan Savage, Neal Cardwell, David Wetherall, e Tom Anderson. TCP Congestion Control with a Misbehaving Receiver. *Computer Communication Review*, 29(5), 1999. Disponível em `http://citeseer.nj.nec.com/savage99tcp.html`.

[35] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 2nd edition, 1996.

[36] Christoph L. Schuba, Ivan V. Krsul, Markus G. Kuhn, Eugene H. Spafford, Aurobindo Sundaram, e Diego Zamboni. Analysis of a Denial of Service Attack on TCP. *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, páginas 208–223. IEEE Computer Society Press, maio de 1997. Disponível em `https://www.cerias.purdue.edu/techreports-ssl/public/97-06.ps`.

[37] Snort Project. Snort network intrusion detection system, version 1.9.0, 2002. Disponível em `http://www.snort.org`.

[38] Tcpdump Project. tcpdump sniffer, version 0.6.2, 2002. Disponível em `http://www.tcpdump.org`.

[39] Bennett Todd. Distributed Denial of Service Attacks, fevereiro de 2000. Disponível em `http://www.opensourcefirewall.com/ddos_whitepaper_copy.html`.

[40] Yuri Volobuev. Playing redir games with ARP and ICMP, setembro de 1997. Disponível em `http://bugtraq.inet-one.com/dir.1997-09/msg00057.html`.

[41] P. Weiss. Yellow Pages Protocol Specification, Sun Microsystems, Inc. Technical Report, 1985.

[42] David A. Wheeler. *Program Library HOWTO*, agosto de 2002. Disponível em `http://www.dwheeler.com/program-library/Program-Library-HOWTO/index.htm%l`.

[43] Ira S. Winkler e Brian Deal. Information Security Technology? Don't Rely on It. A Case Study in Social Engineering. *5TH USENIX UNIX Security Symposium*, 1995. Disponível em `http://www.usenix.org/publications/library/proceedings/security95/winkl%er.html`.