Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Webspy: uma aplicação de monitoramento Web em tempo real

André Figueira Lourenço

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. MSc. João José Costa Gondim

Brasília
2013

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Díbio Leandro Borges

Banca examinadora composta por:

Prof. MSc. João José Costa Gondim (Orientador) — CIC/UnB
Prof. Dr. Diego de Freitas Aranha — CIC/UnB
Prof. Dr. Robson Albuquerque — ENE/UnB

CIP — Catalogação Internacional na Publicação

Lourenço, André Figueira.

Webspy: uma aplicação de monitoramento Web em tempo real / André Figueira Lourenço. Brasília : UnB, 2013.
147 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2013.

1. monitoramento *Web*, 2. ARP *Spoofing*, 3. *man-in-the-middle*, 4. SSL *Stripping*

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Webspy: uma aplicação de monitoramento Web em tempo real

André Figueira Lourenço

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. MSc. João José Costa Gondim (Orientador)
CIC/UnB

Prof. Dr. Diego de Freitas Aranha
CIC/UnB

Prof. Dr. Robson Albuquerque
ENE/UnB

Prof. Dr. Díbio Leandro Borges
Coordenador do Bacharelado em Ciência da Computação

Brasília, 25 de julho de 2013

Agradecimentos

Agradeço a todos parentes e amigos que acreditaram e confiaram na minha capacidade e que compreenderam os esforços necessários para a conclusão deste trabalho.

Resumo

O surgimento e difusão da *Web* permitiu a criação de diversas aplicações, entre elas as domésticas, comerciais e móveis. Dessa forma, o uso da *Web* através da Internet tornou-se uma ferramenta muito versátil e poderosa. Porém, como não é possível garantir que todos que a utilizam são bem intencionados, discute-se cada vez mais a necessidade de ferramentas de monitoramento de seu uso. Este trabalho busca apresentar uma aplicação para monitoramento do uso da *Web* em tempo real com foco em um *host* específico de uma rede: o *Webspy*. Essa aplicação foi desenvolvida como prova de conceito de que é possível realizar o monitoramento de tráfego da *Web* através de técnicas de ataque do tipo *man-in-the-middle*, em especial a técnica de ARP *Spoofing*. Dentre as funcionalidades propostas destacam-se a filtragem e visualização de tráfego relativo a páginas da *Web* em tempo real e a reprodução da visualização das páginas interceptadas na ordem em que foram acessadas pelo *host* monitorado. Todos os conceitos e técnicas que possibilitam o desenvolvimento da aplicação são apresentados e discutidos. A implementação final da aplicação apresentou resultados muito positivos quanto à visualização de páginas HTTP, conseguindo cumprir os objetivos propostos com pequenas limitações. A versão final da aplicação incluiu também a funcionalidade de visualização de páginas HTTPS utilizando a técnica de SSL *Stripping*.

Palavras-chave: monitoramento *Web*, ARP *Spoofing*, *man-in-the-middle*, SSL *Stripping*

Abstract

The rise and diffusion of the Web has enabled the development of several applications: domestic, commercial and mobile. Like so, the use of the Web through the Internet has become a powerful and versatile tool. However, since it's not possible to guarantee that everyone who use this tool is well intentioned, the need for monitoring tools is a topic being discussed more and more often. This work presents an application called Webspy that is capable of monitoring the use of the Web associated with a specific host in a computer network. This application was developed as a proof of concept that it is possible to accomplish Web monitoring through man-in-the-middle atack techniques, namely ARP Spoofing. The following features are emphasized: filtering and visualization of traffic related to Web pages in real time and the replay of the visualizations of intercepted pages in the order they were viewed by the monitored host. All concepts and techniques that make this approach possible are presented and discussed in this work. The final implementation of the application revealed impressive results in regards to the visualization of HTTP pages, fullfiling the proposed objectives with litle limitations. The final version also included the feature of viewing HTTPS pages through the use of the technique known as SSL Stripping.

Keywords: Web monitoring, ARP Spoofing, man-in-the-middle

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Desde os primórdios, algumas das principais atividades do homem são: geração, armazenamento e transmissão de informação. O surgimento dos computadores e sua constante evolução possibilitou produzir, armazenar e processar um grande volume de informações de maneira rápida e fácil. Essa capacidade tornou os computadores cada vez mais populares, fazendo com que ocupassem um papel essencial na vida de qualquer pessoa.

Apesar de tantas funcionalidades proporcionadas com a criação dos computadores, eventualmente surgiu a necessidade de transmitir as informações manipuladas por eles. Nesse contexto, surgiram as redes de computadores, sendo a *Internet* a maior delas. A *Internet* é, na verdade, um sistema global e interconectado de redes de computadores que permite a comunicação entre quaisquer computadores que estejam conectados a ela, também chamados de *hosts* ou hospedeiros, como definido por [23].

O novo ambiente criado pela *Internet* propiciou o desenvolvimento de serviços capazes de distribuir e propagar informação como a *Web* (*World Wide Web*), *e-mail*, transferência de arquivos, entre outros, sendo o primeiro o foco deste trabalho. É importante distinguir a *Internet* da *Web*: a primeira é um sistema de redes interligadas, enquanto que a segunda é um sistema de documentos de hipertexto interligados que, por sua vez, podem ser acessados pela *Internet*.

O surgimento e difusão da *Web* permitiram a criação de diversas aplicações, entre elas as domésticas, comerciais e móveis [38]. Dessa forma, o uso da *Web* através da *Internet* tornou-se uma ferramenta muito versátil e poderosa. Porém, como não é possível garantir que todos que a utilizam são bem intencionados, discute-se cada vez mais a necessidade de ferramentas de monitoramento de seu uso.

1.1 Objetivos

Este trabalho busca apresentar uma aplicação para monitoramento do uso da *Web* em tempo real com foco em um *host* específico de uma rede. A aplicação foi denominada *Webspy* em homenagem a uma aplicação homônima presente no pacote *dsniff* desenvolvido por *Dug Song* para plataformas *Linux* [34]. O *Webspy* foi desenvolvido para desempenhar as seguintes funções:

1. Verificar quais máquinas estão conectadas a uma rede e seus endereços.

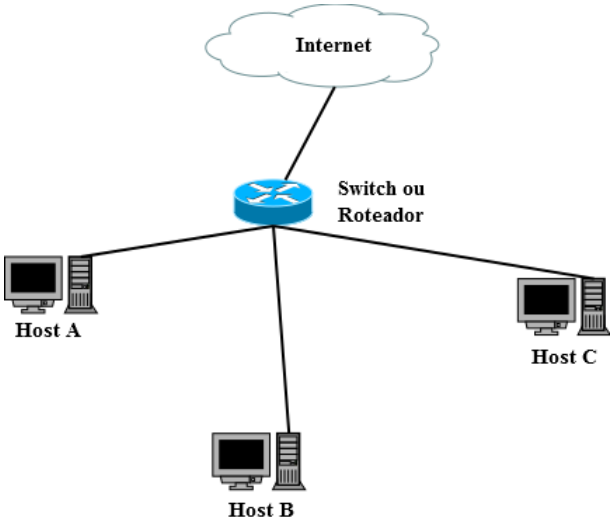


Figura 1.1: Configuração de rede na qual o *Webspy* poderia ser utilizado

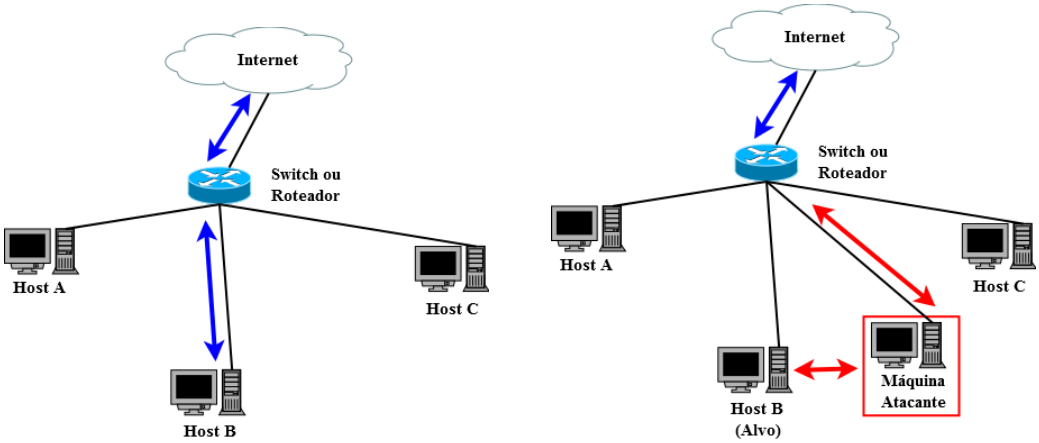
2. Analisar o tráfego de rede que chega e sai de um determinado host da rede (*host* alvo).
3. Interceptar/encaminhar o tráfego destinado ao *host* alvo sem que ele perceba.
4. Filtrar o tráfego destinado ao acesso/utilização de páginas *Web*.
5. Exibir as páginas da *Web* acessadas pelo navegador *Web* da máquina alvo no navegador da máquina atacante em tempo real.

A Figura 1.1 demonstra o contexto de aplicação do *Webspy*: uma rede com um *switch* ou roteador com conexão para a *Internet* e um conjunto de máquinas conectadas a esse dispositivo por meio de cabos. A Figura 1.2a, por sua vez, representa como ocorre o tráfego de rede para o *host* B (que será o dispositivo alvo) antes da inserção da máquina que executa o *Webspy*, enquanto que a Figura 1.2b apresenta como o tráfego será interceptado e redirecionado durante a execução da aplicação.

As técnicas aqui apresentadas, discutidas e implementadas têm finalidade exclusivamente acadêmica e buscam demonstrar, como prova de conceito, vulnerabilidades já conhecidas. Acredita-se que este trabalho possa estimular e orientar a criação de novas (e mais eficazes) formas de proteção. O autor e seu orientador não se responsabilizam por qualquer uso indevido, inadvertido ou inadequado resultantes da aplicação dos conceitos, técnicas e resultados aqui expostos, em especial qualquer uso que não se enquadre no aceitável da prática acadêmica. Os artefatos de *software* produzidos neste trabalho ficam sob a guarda exclusiva do autor e de seu orientador.

1.2 Metodologia

Para o desenvolvimento deste trabalho foram realizadas diversas atividades. A primeira consistiu na revisão teórica dos conceitos básicos de rede e técnicas de ataque do tipo *man-in-the-middle* [17, 22] também conhecidas como *intruder-in-the-middle*. Uma



(a) Tráfego normal (antes da inserção da máquina atacante)

(b) Tráfego redirecionado pela aplicação

Figura 1.2: Tráfego de rede do *host* B antes e durante a utilização do *Webspy*

vez consolidados os conhecimentos, dividiu-se cada uma das funcionalidades da aplicação em módulos e foram implementados protótipos relativos a cada módulo. Após a realização de testes individuais nos protótipos, eles foram reunidos em uma única aplicação que foi testada em um ambiente de rede controlado.

1.3 Organização do trabalho

No capítulo 2, são apresentados os principais conceitos das redes de computadores, bem como a evolução dos dispositivos que permitiram criá-las, ressaltando que à medida que ficaram mais desenvolvidos, tornaram o monitoramento uma tarefa cada vez mais difícil.

Os dois capítulos seguintes abordam conceitos teóricos que fundamentam a implementação das principais funcionalidades do *Webspy*. O capítulo 3 apresenta técnicas de descoberta de rede e discute os conceitos de ARP *Spoofing* e *man-in-the-middle*. Por sua vez, o capítulo 4 explica como é feita a filtragem e visualização do tráfego da *Web*.

O capítulo 5 discute decisões e detalhes de implementação, apresentando a arquitetura do *Webspy* e os resultados obtidos. Por fim, o capítulo 6 expõe considerações finais e discute trabalhos futuros.

Capítulo 2

Rede e dispositivos de rede

Uma rede de computadores pode ser definida como uma infraestrutura de *hardware* e *software* que permite o compartilhamento de recursos e a troca de informações entre os dispositivos que a compõem (chamados de hospedeiros, *hosts* ou nós). As redes podem ser classificadas quanto ao seu tamanho e abrangência como mostra a Tabela 2.1.

Uma rede local também chamada de LAN (*Local Area Network*) é uma rede que se estende em média a um andar ou um único edificio. Uma MAN (*Metropolitan Area Network*) consiste em uma rede que interconecta diversas LANs que se encontram numa mesma região geográfica como, por exemplo, uma cidade. Uma WAN (*Metropolitan Area Network*), por sua vez, refere-se a uma rede que abrange países ou continentes e a *Internet*, por fim, representa a rede que tem a capacidade de abragência máxima, ou seja, o planeta inteiro.

2.1 *Software* de Rede

Uma rede deve possibilitar a comunicação entre todos os nós que fazem parte dela. Do ponto de vista de *software*, foi preciso estabelecer padrões que definem como cada nó será identificado na rede, como serão divididas as informações transmitidas, qual e quando cada nó poderá iniciar uma transmissão. Esses padrões constituem um **protocolo de rede** que, conforme [38], define o formato e a ordem das mensagens trocadas entre duas ou mais entidades comunicantes, bem como as ações realizadas na transmissão e/ou recebimento de uma mensagem ou outro evento.

Tabela 2.1: Redes classificadas por sua abrangência.

Tipo de Rede	Abrangência Média (distância de processadores interconectados)
LAN	10 m a 1 km
MAN	10 km
WAN	100 km a 1000 km
Internet	10000 km

Para reduzir a complexidade do projeto, o **software de rede** foi dividido em diversos protocolos de rede que são organizados como uma pilha de camadas ou níveis. Essa organização permite que cada camada tenha um propósito específico e somente precise saber lidar com as camadas diretamente abaixo e acima. Vários modelos foram criados para padronizar como deveria ser a estrutura de camadas, sendo o modelo OSI (*Open Systems Interconnection*) e o TCP/IP os dois modelos mais conhecidos.

O modelo OSI possui 7 camadas como definido em [45]:

1. Física: camada de mais baixo nível do modelo. Define especificações elétricas e físicas dos dispositivos de rede para que possa ser realizada a transmissão de *bits* brutos através de um canal de comunicação (isto é, possibilita a transformação dos dados digitais em sinais que podem ser transmitidos por um canal).

2. Enlace: o principal papel dessa camada é transformar um canal de transmissão bruta em um meio livre de erros de transmissão detectando e corrigindo-os. Esta camada também se responsabiliza pelo endereçamento físico, controle de acesso e controle de fluxo.

3. Rede: provê transferência de sequências de dados de tamanho variável entre *hosts* de diferentes redes. Nessa camada, é definido um novo esquema de endereçamento (desta vez lógico ao invés de físico), tendo em vista que o esquema elaborado na camada de enlace permite apenas a transferência de dados entre *hosts* de uma mesma rede.

4. Transporte: provê transferência de dados transparente e confiável entre usuários finais para as camadas superiores. O modelo OSI define 5 classes que podem ser implementadas por uma camada de transporte. Cada classe apresenta um conjunto diferente de papéis e serviços disponibilizados, como controle de fluxo, mecanismo de retransmissão em *timeout*, recuperação de erros entre outros.

5. Sessão: controla as sessões que são estabelecidas entre os computadores. Uma sessão oferece diversos serviços, como sincronização da transmissão, controle de diálogo e gerenciamento de símbolos.

6. Apresentação: torna possível a comunicação entre computadores com diferentes estruturas de representação de dados definindo uma maneira de intercambiar estruturas para chegar a uma estrutura comum que possa ser entendida por ambos participantes do diálogo.

7. Aplicação: provê a principal interface para que programas possam utilizar os serviços de comunicação oferecidos pelas demais camadas.

Já o modelo TCP/IP (também chamado de pilha TCP/IP) é divido em 4 camadas como definido em [31]:

1. Interface de rede (também chamada de camada de enlace): lida com a transmissão de informação (codificada em *bits* e organizada em estruturas denominadas **pacotes**) entre diferentes nós de um mesmo enlace.

2. Inter-rede ou *Internet*: tem o objetivo de possibilitar o envio de pacotes entre nós de diferentes redes, isto é, nós que não se encontram no mesmo enlace.

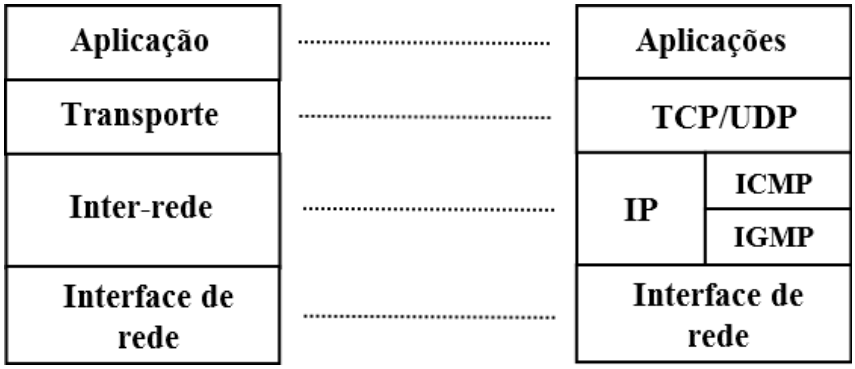


Figura 2.1: Camadas do modelo TCP/IP e protocolos correspondentes

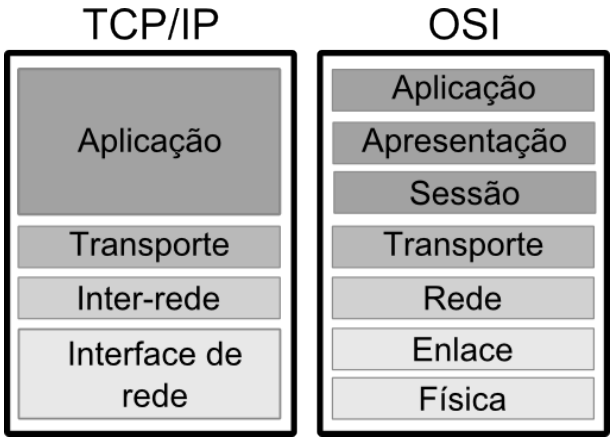


Figura 2.2: Camadas do modelo OSI comparadas com as camadas do modelo TCP/IP.

3. Transporte: provê a transferência de dados ponto a ponto, dessa forma, dois nós podem estabelecer uma conexão entre si.

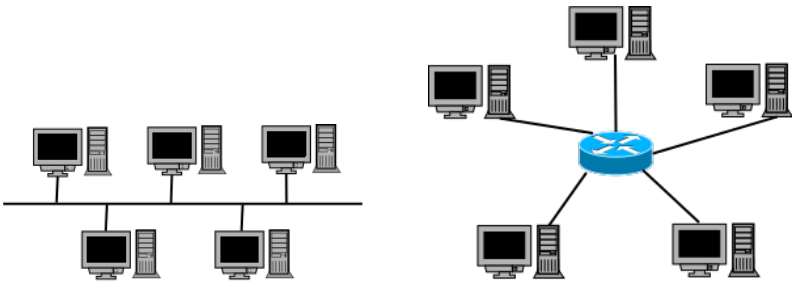
4. Aplicação: corresponde à implementação do programa que deseja se comunicar com outro *host* da rede. Nessa camada são processados os dados recebidos/enviados.

A Figura 2.1 apresenta as camadas do modelo TCP/IP e os protocolos que são utilizados em cada uma delas, enquanto que a Figura 2.2 mostra como o modelo TCP/IP poderia ser mapeado no modelo OSI.

O modelo OSI é muito importante por seu valor teórico e didático, pois foi elaborado com base em diversos princípios aprovados e validados pela ISO (*International Standards Organization*). No entanto, a criação do modelo TCP/IP e o surgimento de suas primeiras implementações mostraram que um modelo simplificado teria uma implantação mais simples e, portanto, maior alcance.

2.2 *Hardware* de Rede

Do ponto de vista de *hardware*, os computadores que compõem uma rede podem estar organizados de diversas maneiras, representando assim diferentes topologias que possuem



(a) Barramento (b) Estrela

Figura 2.3: Topologias de configuração de rede

vantagens e desvantagens entre si como exemplificado em [43].

Duas das topologias mais conhecidas são barramento e estrela. Em uma rede de barramento, todos os *hosts* da rede estão interconectados por um grande cabo (também chamado de cabo de *backbone*) como mostra a Figura 2.3a. Dessa forma, é preciso controlar como ocorrerá o envio de informação entre *hosts* tendo em vista que todos eles compartilham o mesmo canal para transferência de dados (pacotes). Para isso define-se um padrão sendo que, para esta topologia, o mais conhecido é o **Ethernet** [26] que define uma rede de difusão de barramento com controle descentralizado.

Uma rede Ethernet permite que os *hosts* transmitam informações sempre que desejem, porém caso ocorra uma colisão entre pacotes, os *host* envolvidos aguardam um tempo aleatório e tentam novamente. Esta topologia apresenta diversas vantagens: a instalação é facilitada pois a infraestrutura de *hardware* da rede consiste apenas em uma pequena quantidade de cabos e a falha de um *host* da rede não produz nenhuma consequência para os outros dispositivos conectados a ela. Como desvantagens pode-se citar: o limite do número de *hosts* conectados na rede e a possibilidade de ocorrência de problemas com o cabo *backbone*, que resultaria na interrupção do serviço de rede.

Uma rede com topologia em estrela também opera de acordo com o padrão Ethernet. Nesse esquema de rede, cada *hosts* se conecta a um dispositivo centralizador por meio de um cabo, formando assim uma estrutura em estrela como mostra a Figura 2.3b. O dispositivo centralizador recebe as mensagens de todos outros dispositivos da rede e se encarrega de encaminhá-las aos destinatários pretendidos. Essa estrutura é uma das mais utilizadas, principalmente em redes domésticas, pois a instalação e configuração é fácil e conveniente. No entanto, assim como na topologia em barramento, ainda existe um ponto único de falha: o dispositivo centralizador, também chamado de *dispositivo de rede*.

Um dos primeiros dispositivos de rede criados foi o *hub*. Esse aparelho tem a capacidade de conectar múltiplos dispositivos Ethernet fazendo com que eles formem um único segmento de rede.

O *hub* possui um conjunto de portas Ethernet (de entrada e saída) e funciona como um repetidor de sinal, isto é, tudo que chega à entrada de uma porta é enviado à saída de todas as outras portas como mostra a Figura 2.4. A diferença entre um *hub* e um repetidor de sinal é que o repetidor, por definição, possui somente duas conexões, ou seja, conecta dois dispositivos apenas.

A simplicidade e eficiência (para redes "pequenas") deste dispositivo o tornou muito barato e consequentemente muito difundido. Além disso, seu uso limitava as possíveis

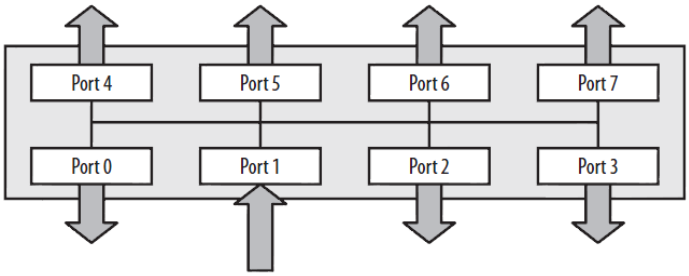


Figura 2.4: Funcionamento de um *hub* [16]

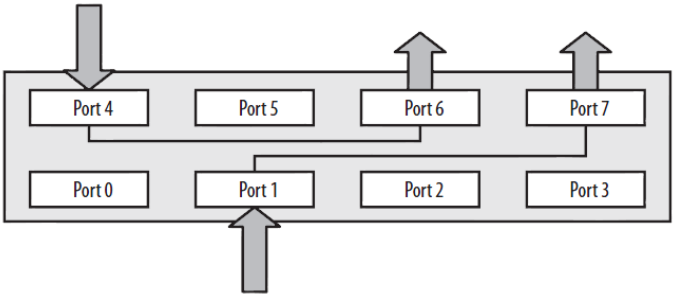


Figura 2.5: Funcionamento de um *switch* [16]

colisões de pacotes a um segmento, isto é, uma seção da rede cujos dispositivos podem se comunicar utilizando endereços de enlace.

Para o propósito visado por este trabalho (monitoramento de uso da *Internet* de um *host* específico de uma rede), esse dispositivo de rede é perfeito, pois bastaria conectar a máquina que deseja realizar o monitoramento no *hub* para que ela pudesse escutar o tráfego da máquina alvo.

No entanto, o *hub* não é um dispositivo de rede escalável, isto é, à medida que a rede cresce, o desempenho é gravemente impactado (pois aumenta-se a probabilidade de ocorrência de colisões), além de que atualmente são considerados obsoletos.

Isso porque o *hub* replica e transmite o tráfego de uma porta para todas as outras mesmo quando é necessário transmitir mensagens para um único *host*. Com intuito de resolver esse e outros problemas, foi desenvolvida a evolução do *hub*, o dispositivo conhecido como *switch*.

O *switch* é um dispositivo de rede capaz de dividir uma rede em pequenos pedaços lógicos também chamados de segmentos, como definido em [14]. Esse dispositivo possui um esquema de portas parecido com o do *hub*, porém seu funcionamento é muito diferenciado.

O funcionamento do *switch* está descrito em detalhes em [33]. Basicamente, ele analisa o tráfego que chega em uma porta de entrada e determina para qual porta de saída ele deverá ser transmitido, portanto opera sobre a camada 2 (enlace). Além disso, esse aparelho registra quais dispositivos estão conectados em cada porta, o que permite realizar um processamento muito mais eficaz.

A Figura 2.5 exemplifica o funcionamento de um *switch* considerando a transmissão de dados do dispositivo conectado na porta 1 para o conectado na porta 7 (bem como a transmissão do dispositivo da porta 4 para o da porta 6).

Outra distinção que deve ser realizada entre um *hub* e um *switch* é que o primeiro atua somente no nível de hardware (camada 1), enquanto que o último opera em conjunto com um software, uma implementação do protocolo Ethernet, o que caracteriza-o como um dispositivo de camada 2 (enlace), conforme explicitado anteriormente.

Apesar de apresentar uma circuitaria muito mais avançada que a dos *hubs*, os *switches* rapidamente tomaram seu lugar como principal dispositivo de rede tanto para redes domésticas quanto para redes de maior porte.

Hubs e *switches* são dispositivos muito úteis para conectar *hosts* dentro de um mesmo enlace ou sub-rede, porém não permitem conectar *hosts* que estão em sub-redes distintas.

Para que ocorra uma conexão desse tipo, é preciso adotar uma forma de identificação para cada nó, de maneira que ela seja válida entre sub-redes e não mais somente em uma única sub-rede. Para suprir essa necessidade foi criado o roteador, um dispositivo de rede capaz de entender e utilizar o protocolo IP (*Internet Protocol*), operando assim como um dispositivo de camada 3 (rede).

Um roteador pode ser descrito como um dispositivo de rede que encaminha pacotes de dados entre redes de computadores, criando assim uma interconexão entre sub-redes. Para isso, esse dispositivo gerencia endereços e rotas, isto é, caminhos para se chegar a um determinado endereço.

Capítulo 3

Descoberta de Rede e ARP *Spoofing*

A expressão descoberta de rede refere-se à atividade de descoberta de quais dispositivos estão conectados a uma rede e onde eles estão, isto é, a partir de qual endereço eles podem ser encontrados. A descoberta (também chamada de varredura) pode ser realizada em diferentes níveis devendo-se basear no protocolo correspondente ao nível escolhido.

No nível da camada de enlace, é possível realizar a varredura ARP (*Address Resolution Protocol*) também chamada de ARP *sweep*. Cada dispositivo conectado a uma sub-rede deve possuir uma placa de rede que contém um endereço denominado endereço MAC (*Media Access Control*).

Esse endereço também chamado de endereço de máquina ou de *hardware* é único e é responsável por identificar cada máquina dentro de uma mesma sub-rede. Um endereço MAC possui 48 *bits* (tipicamente representados por 6 pares de números hexadecimais separados por dois pontos, por exemplo: AA:AA:AA:AA:AA:AA), o que permite gerar aproximadamente 2^{48} combinações diferentes. Esse número é mais que suficiente para suportar a quantidade de dispositivos que necessitam acessar uma rede (e/ou a *Internet*) considerando que esse endereço é utilizado apenas para identificação em uma sub-rede.

Cada dispositivo conectado a uma rede possui também um endereço cuja função é identificar *hosts* de diferentes redes, como definido pelo protocolo IP. Esse endereço possui 32 *bits* (considerando a versão IPv4) geralmente representado como um conjunto de 4 números de 0 a 255 separados por pontos, como por exemplo 192.168.1.1.

O endereço de rede é dividido em *netid*, porção que identifica uma rede e *hostid*, porção que identifica um *host*. O número de *bits* alocados para cada divisão permite uma distribuição de endereços adequada para o tamanho de cada rede.

Com este intuito, os endereços foram divididos em classes. A Tabela 3.1 exibe o número de combinações possíveis de redes e *hosts* para 3 das classes de endereçamento mais utilizadas em LANs e a Figura 3.1 mostra como estão divididas cada classe.

Durante os cálculos de número de redes e hosts, é importante lembrar que devem ser subtraídos dois endereços que são reservados: o endereço rede/sub-rede (todos os *bits* do *hostid* em 0) e o endereço de *broadcast* (todos os *bits* do *hostid* em 1).

3.1 Protocolo ARP

O protocolo ARP provê um método de mapeamento dinâmico capaz de traduzir um endereço de rede em um endereço de máquina correspondente. De acordo com [35], o

Tabela 3.1: Classes de endereçamento IPv4

Classe	Nº de Redes	Nº de hosts	Notação
A	$2^8 - 2 = 126$	$2^{24} - 2 = 16777214$	/8
B	$2^{16} - 2 = 126$	$2^{16} - 2 = 65534$	/16
C	$2^8 - 2 = 126$	$2^8 - 2 = 254$	/24

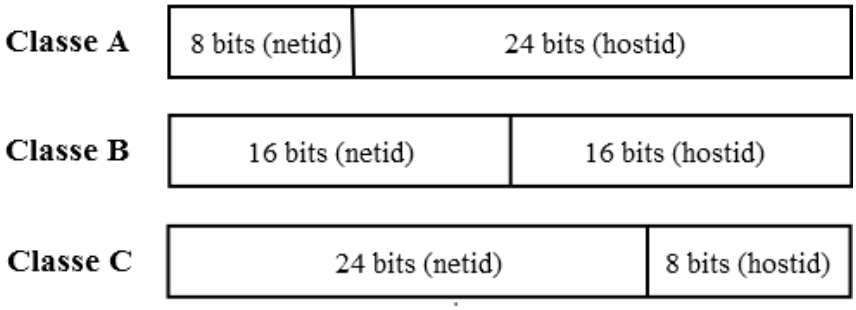


Figura 3.1: Divisões de *bits* para cada classe de endereçamento

termo dinâmico refere-se ao fato de que essa tradução ocorre automaticamente sem que o usuário final ou o administrador do sistema precise tomar alguma atitude específica.

Este protocolo é executado sob a camada 2 (enlace) e, portanto, os dados transmitidos devem apresentar o cabeçalho Ethernet, constituindo assim um *frame* como mostra a Figura 3.2. Existem diversos formatos para o cabeçalho Ethernet, portanto será considerado, neste trabalho, o padrão proposto em [20].

Dessa forma, o cabeçalho deve possuir 18 *bytes*: 12 *bytes* para os endereços de máquina de origem e destino, 2 *bytes* para o tipo de protocolo correspondente aos dados, 46 a 1500 *bytes* para os dados a serem transmitidos e 4 *bytes* para o CRC, campo que possibilita a verificação de integridade.

Todo dispositivo conectado à rede possui uma tabela que contém uma lista de mapeamentos de endereços denominada *cache* ARP. Essa lista, contém apenas os mapeamentos mais recentes e cada entrada possui um tempo de expiração (normalmente 20 minutos) após o qual ela é removida.

Para que um *host* A envie uma mensagem para um *host* B, é preciso que o primeiro saiba o endereço de rede do segundo. Para isso, ele procura em seu *cache* ARP por uma entrada com o endereço de rede de B. Se houver, ele recupera o endereço de máquina correspondente e envia a mensagem encapsulando-a em um *frame* Ethernet.

Caso não exista, ele envia uma mensagem de requisição ARP (ARP *request*) para

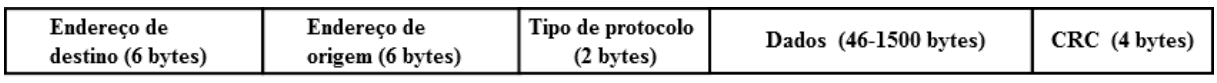
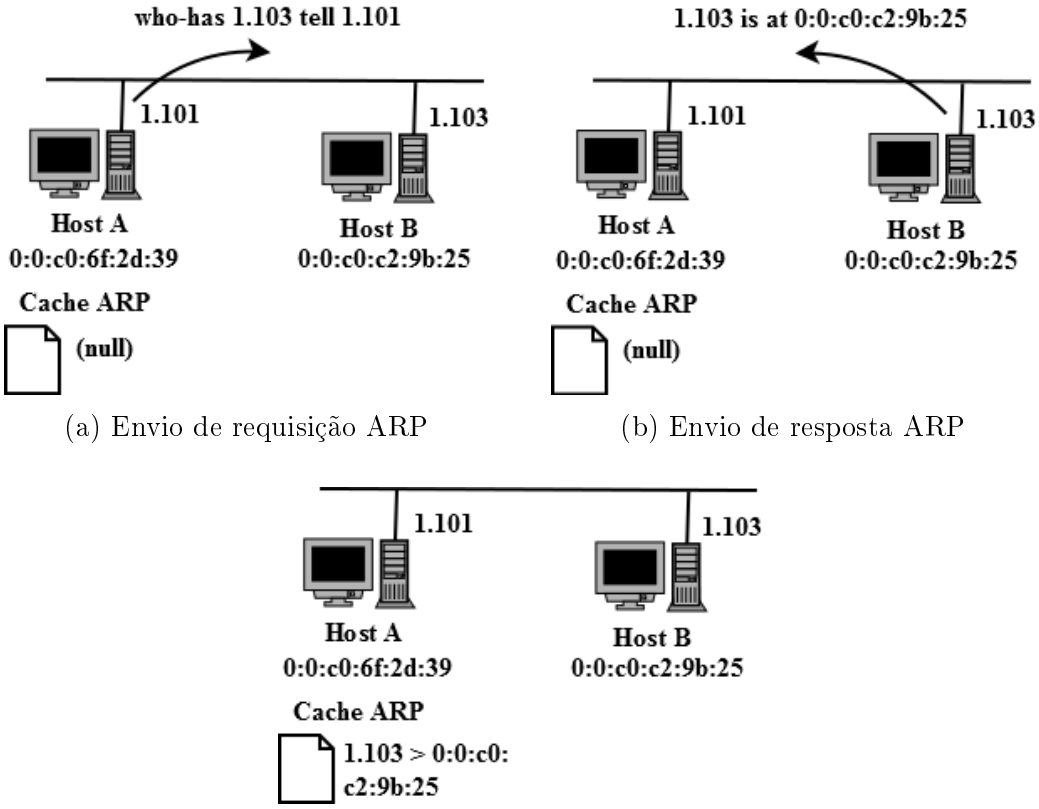


Figura 3.2: *Frame* Ethernet como definido em [20]



(a) Envio de requisição ARP

(b) Envio de resposta ARP

(c) Atualização de *cache* ARP

Figura 3.3: Diálogo ARP para que um *host* A encontre o endereço de máquina de um *host* B

todos os *hosts* de sua sub-rede (utilizando o endereço de máquina de *broadcast*). Essa mensagem, também conhecida como *who-has*, contém duas informações: o endereço de rede para o qual se quer saber o endereço de máquina e o endereço de rede da máquina que faz a requisição.

Quando o *host* B recebe a requisição, ele envia uma resposta ARP (ARP *reply*) para A (B sabe o endereço de máquina de A, pois ele está presente no *frame* da requisição ARP como endereço de origem). Esta mensagem, por sua vez, é conhecida como *is-at* e também contém duas informações: o endereço de rede procurado e seu endereço de máquina correspondente. Ao receber a resposta, o *host* A adiciona uma entrada correspondente ao *host* B em seu *cache* ARP. A Figura 3.3 ilustra esse processo.

Uma característica fundamental e necessária para a realização do ataque discutido neste trabalho é a volatilidade do protocolo ARP. Sempre que um *host* recebe uma resposta ARP, ele atualiza seu *cache*, portanto, um *host* malicioso pode facilmente injetar informações arbitrárias no *cache* de um determinado *host*.

O protocolo ARP permite também a realização do ARP *sweep*. Para realizar este tipo de varredura, consideram-se todos os endereços válidos de uma rede. Por exemplo, caso a rede na qual será feita a varredura seja uma rede de classe C, sabe-se que ela possuirá no máximo 254 dispositivos conectados.

A varredura consiste em injetar, na rede, requisições ARP para cada endereço de rede disponível. Dessa forma, se, ao se conectar em uma rede de classe C, o dispositivo que

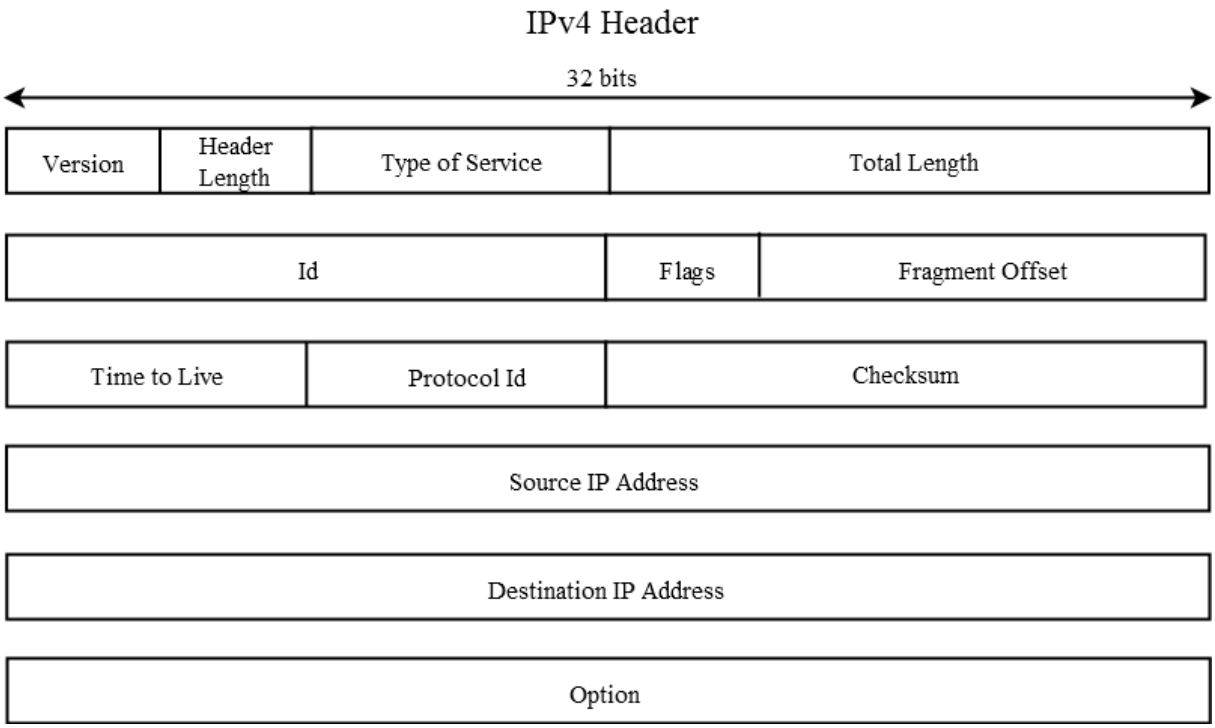


Figura 3.4: Campos do cabeçalho IPv4 como definido em [21]

deseja realizar a varredura recebe o endereço IP 192.168.1.101, ele irá enviar requisições ARP para os endereços de 192.168.1.1 a 192.168.1.254.

Em seguida, ele irá receber as respostas ARP dos dispositivos que estão de fato conectados à rede. Uma vantagem desse método é que, ao receber as respostas, o dispositivo que realizou a varredura saberá tanto o endereço de rede quanto o endereço de máquina de cada dispositivo.

3.2 Protocolo ICMP

O protocolo ICMP pode ser considerado uma parte do protocolo IP, portanto, refere-se à camada 3 (rede). Este protocolo tem por objetivo permitir que *gateways* (roteadores) e *hosts* possam trocar mensagens de erro e controle.

As mensagens ICMP são transmitidas utilizando o protocolo IP, isto é, são encapsuladas em um datagrama IP. Para isso, é preciso adicionar o cabeçalho IP às mensagens. Este, por sua vez, é constituído pelos campos mostrados na Figura 3.4 e explicados na Tabela 3.2.

Uma mensagem ICMP possui tamanho variável de acordo com o tipo de mensagem, porém os primeiros 4 *bytes* possuem o mesmo formato para todos tipos: 1 *byte* para o tipo da mensagem, 1 *byte* para o código e 2 *bytes* para o *checksum* (que permite a verificação de integridade da mensagem).

A Tabela 3.3 mostra algumas das diferentes mensagens ICMP e classifica-as em mensagens de consulta (mensagens que pedem/apresentam alguma informação) e mensagens de erro (mensagens que indicam alguma condição na qual foi verificada um erro).

Tabela 3.2: Descrição dos campos do cabeçalho IPv4

Campo	Tamanho	Descrição
Version	4 *bits*	versão do protocolo IP (ex: IPv4 ou IPv6).
Header Length	4 *bits*	tamanho do cabeçalho incluindo os campos de opções.
Type of Service	8 *bits*	especifica características desejadas durante a transmissão do pacote (ex: máximo *throughput*, máxima confiabilidade).
Total Length	16 *bits*	tamanho total do datagrama.
Id	16 *bits*	número de identificação do datagrama.
Flags	3 *bits*	indicam condições especiais (ex: *Don't Fragment*, *More fragments*).
Fragment Offset	13 *bits*	indica qual parte da mensagem completa corresponde aquele datagrama (caso tenha ocorrido fragmentação).
Time to Live (TTL)	8 *bits*	limite superior de roteadores pelos quais o datagrama deve poder passar.
Protocol Id	8 *bits*	identifica o protocolo de camada imediatamente superior que passou dados que serão encapsulados pelo IP.
Checksum	16 *bits*	permite realizar verificações de integridade do datagrama.
Source IP Address	32 *bits*	endereço de origem do datagrama.
Destination IP Address	32 *bits*	endereço de destino do datagrama.
Option	variável	armazena informações opcionais do datagrama.

Tabela 3.3: Mensagens ICMP - adaptada de [35]

Tipo	Código	Descrição	Consulta	Erro
0	0	*echo reply*	•	
3		*destination unreachable*:		
	0	*network unreachable*		•
	1	*host unreachable*		•
	2	*protocol unreachable*		•
	3	*port unreachable*		•
	4	*fragmentation needed but don't fragment bit set*		•
	5	*source route failed*		•
⋮	⋮	⋮	⋮	⋮
8	0	*echo request*	•	
9	0	*router advertisement*	•	
10	0	*router solicitation*	•	
11		*time exceeded*		
	0	*time to live equals 0 during transit*		•
	1	*time to live equals 0 during reassembly*		•
⋮	⋮	⋮	⋮	⋮

Uma das preocupações do protocolo ICMP é a ocorrência de *broadcast storms*, isto é, a geração de um grande número de mensagens de *broadcast* causando a saturação da rede e consumo de toda a banda disponível. Para evitar essa condição, o protocolo adota um conjunto de regras que impedem a geração de mensagens de erro ICMP em resposta a:

1. Uma outra mensagem de erro ICMP.
2. Um datagrama encapsulado por um *frame* com endereço de destino de *broadcast* (isto é, *brodcast* de camada 2).
3. Um datagrama cujo endereço de destino não identifique um único *host* (endereço de loopback, *multicast* ou *broadcast*).
4. No caso de um datagrama fragmentado, qualquer fragmento que não seja o primeiro.

Além disso, quando uma mensagem de erro é gerada, ela sempre carrega o cabeçalho IP e os primeiros 8 *bytes* do datagrama IP que ocasionou o erro, carregando assim mais informações que possam ser utilizadas para verificar a causa do erro. O funcionamento do protocolo ICMP permitiu a criação do programa Ping, uma ferramenta capaz de fornecer informações sobre *hosts* presentes em uma rede.

Basicamente, o Ping injeta, em uma rede, uma mensagem ICMP do tipo *echo request* destinada a um *host* da rede (um endereço de IP) e aguarda um tempo definido para receber uma mensagem de resposta ICMP. A máquina de destino envia uma mensagem ICMP do tipo *echo reply* assim que recebe uma mensagem *echo request*.

Dessa forma, caso receba a mensagem de resposta, a máquina que executou o programa Ping consegue determinar se a máquina de destino está ativa, bem como o *round-trip time*, que pode ser utilizado para estimar a distância relativa entre as duas máquinas.

3.3 Ping *sweep* X ARP *sweep*

Como visto nas seções anteriores, tanto o protocolo ARP quanto o ICMP provêem ferramentas para realizar descoberta de rede. As Figuras 3.5 e 3.6 ilustram como são realizadas cada uma das varreduras.

Pode-se perceber, que a varredura ARP sobrecarrega mais a rede, pois cada mensagem ARP request é enviada em *broadcast*, isto é, é enviada para todos os *hosts* da rede. A varredura Ping, por sua vez, é menos "barulhenta", pois cada mensagem *echo request* é enviada para apenas um *host*, tornando esta varredura mais discreta.

Além disso, o Ping pode ser utilizado para verificar *hosts* ativos e realizar varreduras de outras sub-redes que não a subrede na qual a máquina que executa o programa está conectada.

Isto não pode ser feito com o ARP, pois ele é um protocolo de camada 2 e, portanto, não é roteável. No entanto, muitos roteadores e servidores são configurados para não enviar mensagens *echo reply*, o que inutilizaria a varredura Ping.

Pode-se também, configurar um *firewall* para filtrar mensagens *echo request*. Quanto a essas medidas, a varredura ARP é mais eficiente, tendo em vista que a maioria dos *firewalls* operam apenas em mensagens a partir da camada 3 (rede).

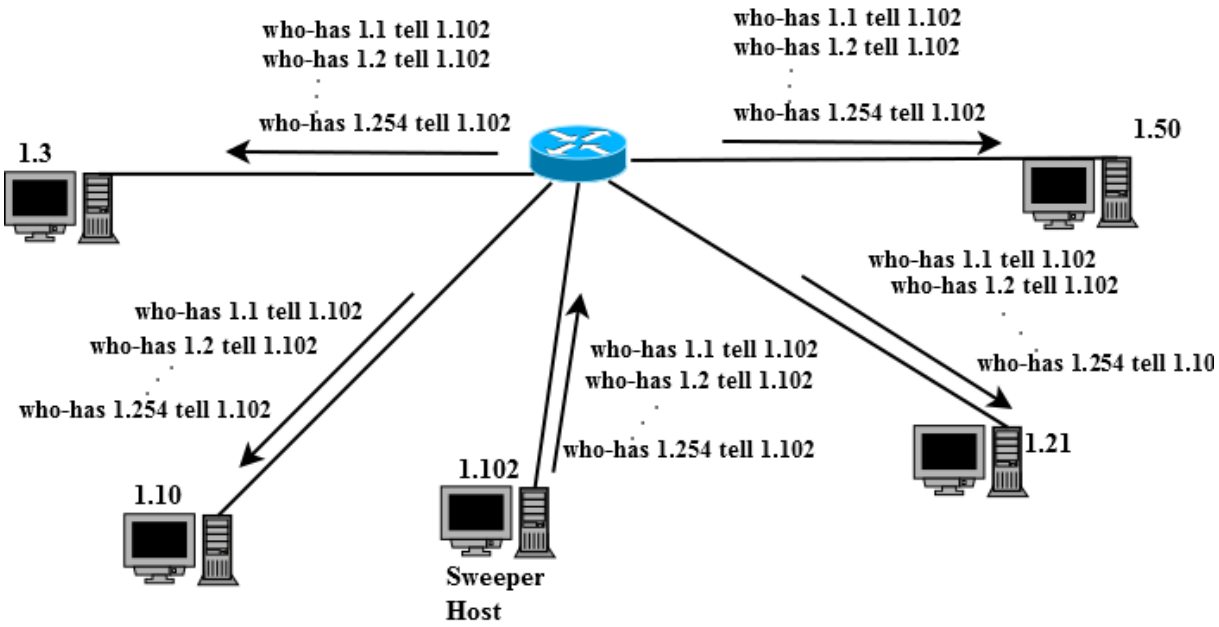
3.4 ARP *Spoofing*

Uma vez realizada a descoberta de rede, sabe-se quais *hosts* estão conectados a uma sub-rede e assim pode-se escolher uma vítima. O próximo passo é interceptar o tráfego de rede da vítima e repassá-lo a ela sem que ela perceba qualquer alteração.

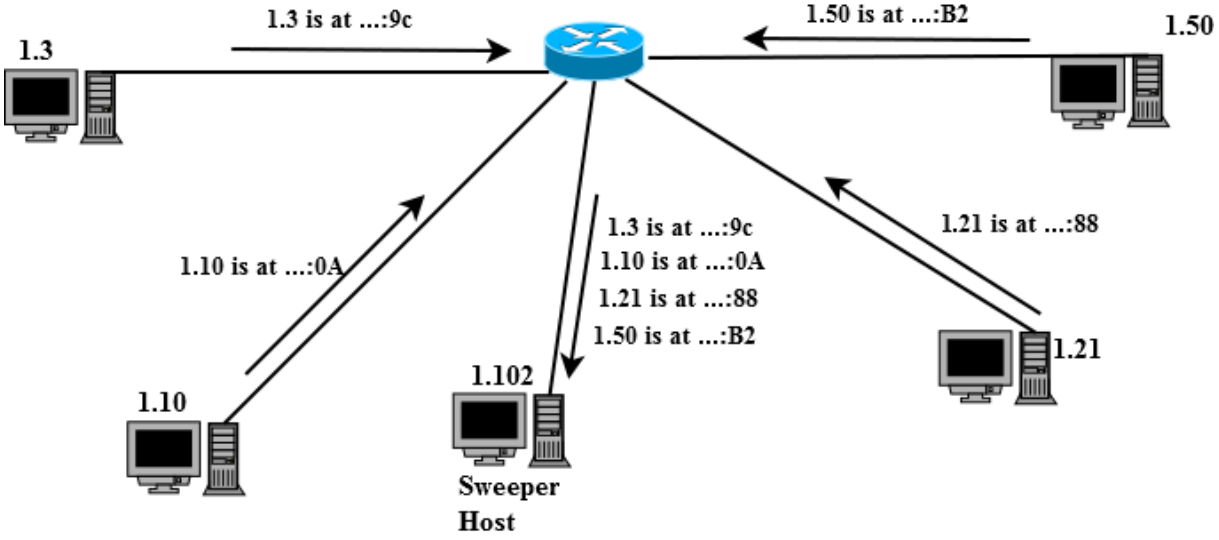
O termo *spoof* refere-se à atividade de falsificar ou forjar alguma informação. Dessa forma, o ARP *Spoofing* consiste na falsificação de mensagens ARP com o objetivo de realizar um ataque do tipo *man-in-the-middle* (MitM) como definido em [40, 24, 11, 12]. Um ataque MitM ocorre sobre um meio de comunicação e tem como alvo duas entidades que mantêm uma comunicação.

A Figura 3.7 ilustra uma conversa entre Alice e Bob antes e durante a realização de um ataque MitM. Uma vez estabelecida a conversa, o atacante, denomidado John, inicia o ataque. John consegue tanto o acesso ao conteúdo da conversa de Alice e Bob, quanto a capacidade de adulterar os dados que passam por ele em qualquer direção. Sendo assim, a realização de um ataque MitM requer que o atacante:

1. Saiba como é realizada a autenticação entre os alvos: caso Alice e Bob possuam um protocolo para garantir que Alice fala realmente com Bob e vice-versa, é preciso que o atacante John seja capaz de forjar informações de maneira a explorar vulnerabilidades desse protocolo - convencendo que Alice está falando com Bob ao falar com John e que Bob está falando com Alice ao falar com John.

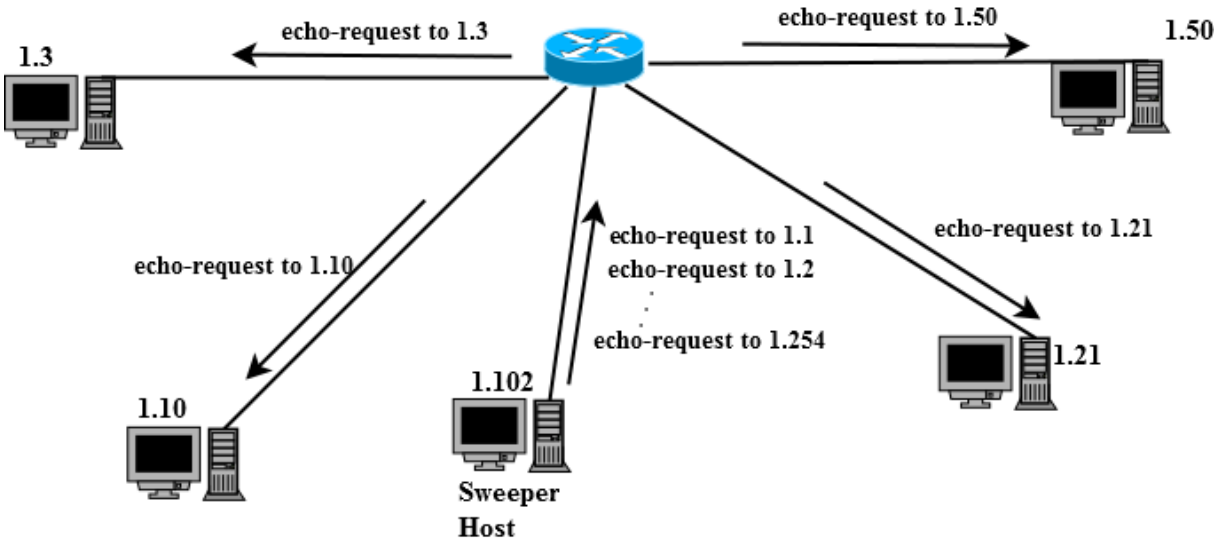


(a) Envio de requisições ARP

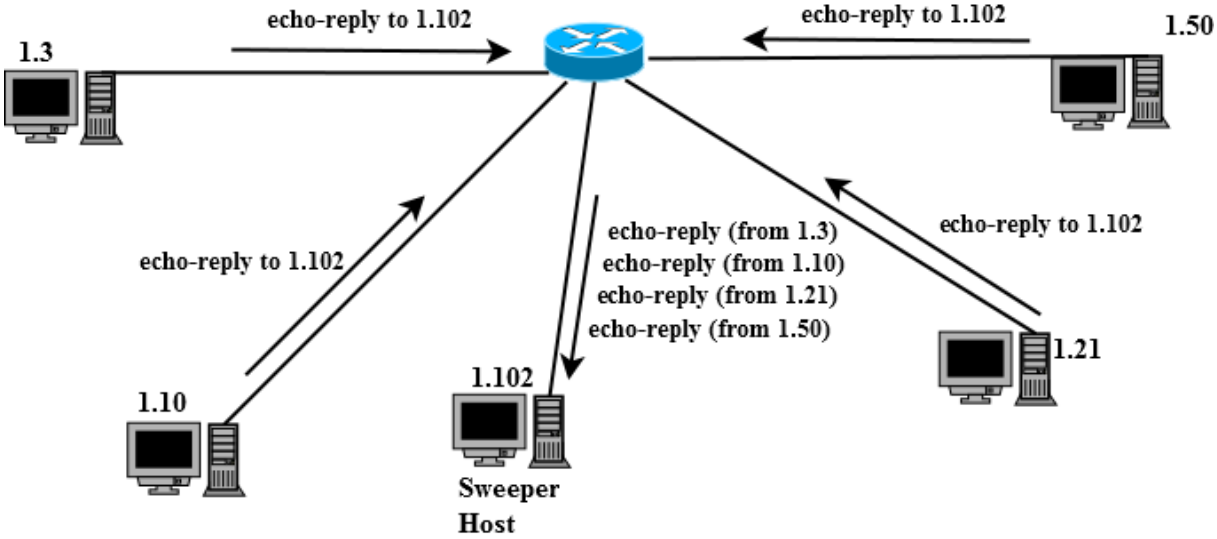


(b) Envio de respostas ARP

Figura 3.5: Varredura ARP

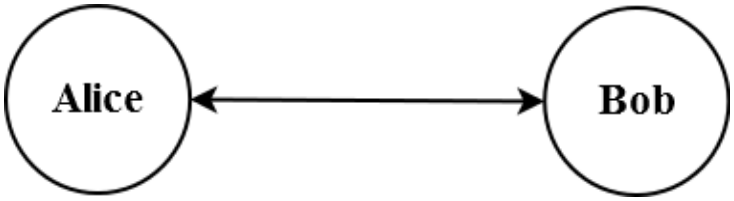


(a) Envio de mensagens *echo request*

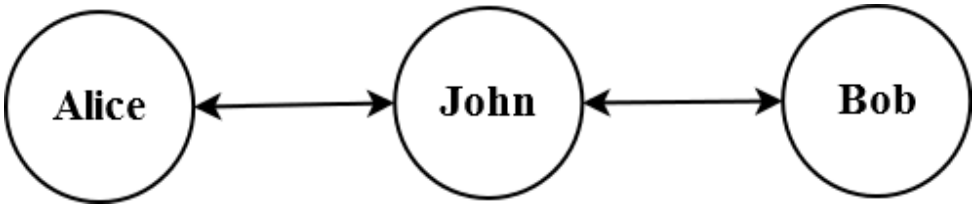


(b) Envio de mensagens *echo reply*

Figura 3.6: Varredura Ping



(a) Conversa entre Alice e Bob antes do ataque



(b) Conversa entre Alice e Bob (por meio de John) após o ataque

Figura 3.7: Exemplo de ataque *man-in-the-middle*

2. Tenha acesso ao canal: o atacante John deve conseguir entrar no meio do canal (como sugere o nome do ataque) de forma que o fluxo de dados que flui de Alice para Bob e vice-versa não tenha como chegar a seu destinatário pretendido sem antes passar por ele.
3. Encaminhe as mensagens: como o atacante entra no meio do canal, passa a ser sua responsabilidade o encaminhamento (ou não, conforme o objetivo do ataque) das mensagens de Alice para Bob e vice-versa.

O ARP *spoofing* tipicamente consiste em duas etapas: o envenenamento ARP (ARP *poisoning*) e o *relay* que serão discutidas nas próximas seções.

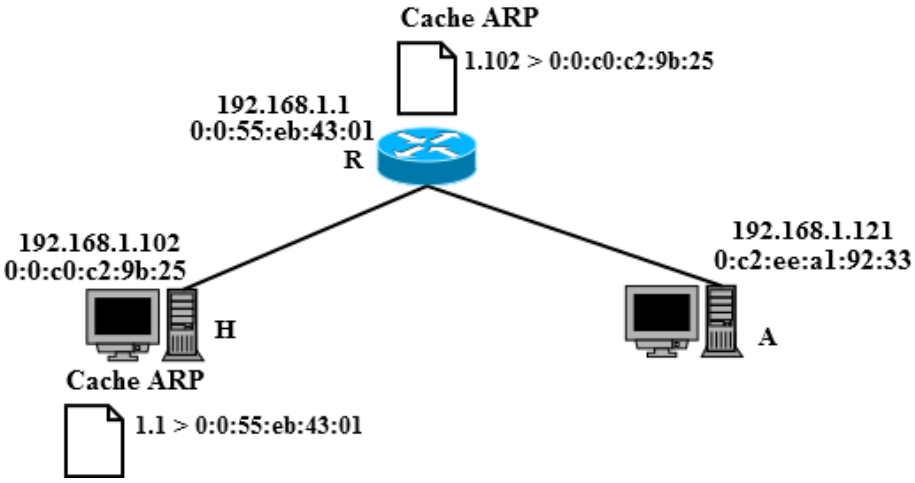
3.4.1 ARP *poisoning*

Seja R um roteador/*switch* e H um *host* conectado à subrede de R, a etapa de envenenamento ARP consiste na realização de *spoofing* de mensagens ARP com o intuito de redirecionar, para uma máquina atacante A, o tráfego que iria de R para H e de H para R.

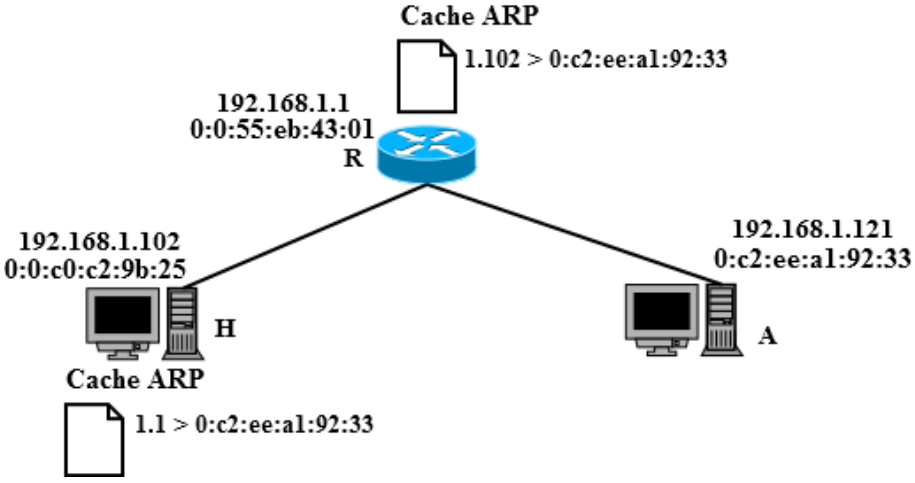
Inicialmente, a subrede possui apenas os *hosts* R e H. Em seguida, o *host* atacante A se conecta à rede e recebe um endereço IP como ilustra a Figura 3.8a. O atacante vai então realizar um método de varredura de *hosts* (como discutido no capítulo 3) na subrede para encontrar uma vítima e seus endereços MAC e IP.

Uma vez conhecidos os endereços da vítima, o atacante envia uma mensagem de resposta ARP para R informando que o IP 192.168.1.102 (IP de R) está associado ao MAC 0:c2:ee:a1:92:33 (MAC de A). Dessa forma, o cache ARP de R é atualizado e todo tráfego com o IP de destino de H vai ser direcionado para A. De forma análoga, o fluxo de H com destino R é redirecionado para A. A Figura 3.8b, ilustra o resultado final.

Por fim, para que o envenenamento ARP seja de fato eficaz, é preciso repetir o processo de acordo com a taxa de expiração das entradas do cache ARP. Isso porque, caso a entrada de um *host* expirasse, aquele *host* iria enviar uma mensagem de requisição ARP e receberia uma resposta legítima, o que reestabeleceria o tráfego normal. Essa repetição,



(a) Cenário anterior ao envenenamento ARP.



(b) Cenário após envenenamento ARP.

Figura 3.8: Exemplo de envenenamento ARP.

acaba facilitando a detecção deste tipo de ataque, pois é gerado um fluxo periódico de respostas ARP como abordado em [41, 29, 12].

3.4.2 *Relay*

Esta etapa, consiste no encaminhamento das mensagens que chegam ao *host* atacante. Há duas maneiras de fazer isso: de forma automática ou de forma manual. Para realizar o processo de forma automática basta habilitar o roteamento de pacotes IP, uma configuração da implementação da pilha TCP/IP que vem desabilitada por padrão nos sistemas operacionais de propósito geral (*Windows* e distribuições *Linux*). Apesar de prática, essa maneira oferece menos poder sobre os pacotes que estão sendo roteados, pois a pilha TCP/IP se encarrega de tudo.

A outra maneira consiste em realizar manualmente a escuta dos pacotes e o encaminhamento injetando-os de volta na rede (desta vez com o endereço MAC de destino correto). Dessa forma, a aplicação que realiza o ARP *spoofing* deve lidar com a escuta e injeção de pacotes, porém ganha mais flexibilidade quanto aos pacotes encaminhados (pode-se adulterá-los ou simplesmente deixar de enviar algum).

Capítulo 4

Filtragem e visualização do tráfego *Web*

Utilizando o ARP *spoofing* consegue-se acesso ao tráfego de rede entre dois *hosts* de uma sub-rede. O próximo passo é identificar e separar a parte desse fluxo de dados que refere-se à navegação de páginas da *Web*. Para isso, é preciso primeiramente entender um pouco mais sobre o protocolo de aplicação que permite a comunicação entre dispositivos e a *Web*.

4.1 Protocolo HTTP

O protocolo HTTP (*Hypertext Transfer Protocol*) é o protocolo de aplicação responsável por definir como se dá a transmissão de dados entre os usuários da *Web*. Ele popularizou a transmissão de dados formatados na linguagem HTML (*Hypertext Markup Language*), além de diversos outros formatos como definido em [15].

O conteúdo da *Web* está distribuído em diversos servidores *Web* que operam de acordo com o protocolo HTTP. Dessa maneira, esses dados podem ser obtidos pelos clientes por meio de requisições HTTP. Esse tipo de arquitetura é conhecido como cliente/servidor, como ilustra a Figura 4.1. O tipo mais comum de cliente HTTP são os navegadores *Web* (*browsers*).

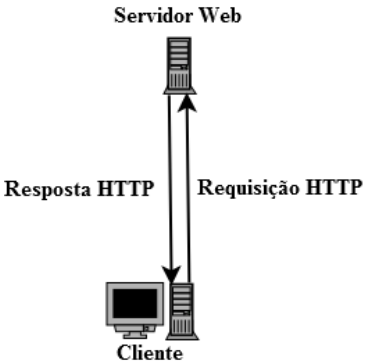


Figura 4.1: Ilustração do modelo cliente/servidor

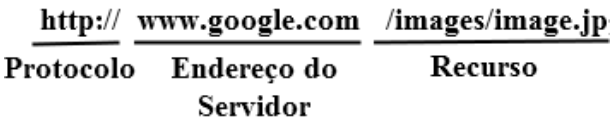


Figura 4.2: Informações contidas em uma URL

4.1.1 Recursos

Cada servidor *Web* possui um conjunto de recursos, que podem ser qualquer tipo de arquivo estático ou dinâmicamente gerado, como páginas HTML, imagens, arquivos de texto, etc. Os servidores possuem também um nome associado que pode ser utilizado pelos clientes para referenciar os recursos pelos quais se interessam. Esse nome é conhecido como URI (*Uniform Resource Identifier*).

Uma URI, por sua vez, pode ser uma URL (*Uniform Resource Locator*) ou uma URN (*Uniform Resource Name*). Uma URL descreve a localização específica de um recurso em um servidor *Web*, definindo: o protocolo a ser utilizado, o endereço do servidor (também conhecido como *hostname*) e o nome do recurso a ser buscado, como mostra a Figura 4.2. Uma URN, por conseguinte, define um nome único para um recurso, independentemente de onde ele se encontre. As URNs são pouco adotadas e ainda estão em fase experimental, segundo [39].

4.1.2 Transações e Mensagens

O processo de interação entre clientes HTTP e servidores é denominado **transação**. Uma transação consiste em duas mensagens: uma mensagem de requisição e uma de resposta. Mensagens HTTP são compostas de sequências de caracteres separadas por linhas e possuem três regiões:

1. Linha inicial: primeira linha da mensagem. Caso a mensagem seja uma requisição, indica o que deve ser feito pelo servidor, caso seja uma resposta, indica o resultado da resposta.
2. Cabeçalho: coleção de campos que fornecem informações adicionais sobre a mensagem. Cada campo possui um nome e valor separados pelo caractere ":". O cabeçalho sempre termina com uma linha vazia.
3. Corpo: após a última linha do cabeçalho, uma mensagem pode conter qualquer tipo de dado (texto ou binário) constituindo assim o corpo da mensagem.

A Figura 4.3 ilustra as mensagens de uma transação simples. Toda requisição HTTP possui um método que indica ao servidor qual ação ele deve realizar. A Tabela 4.1 apresenta os principais métodos e suas funções. Cada resposta HTTP, por sua vez, possui um *status code*: código numérico de 3 dígitos que indica ao cliente qual foi o resultado da ação requisitada. A Tabela 4.2 descreve os códigos mais comuns.

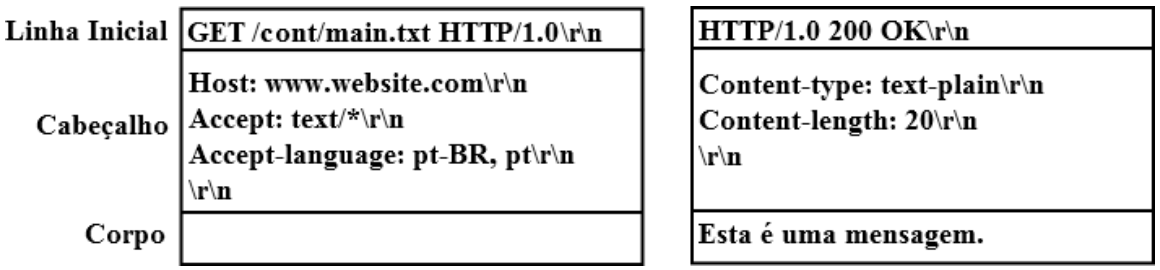


Figura 4.3: Mensagens de uma transação HTTP

Tabela 4.1: Principais métodos do protocolo HTTP

Método	Descrição
GET	solicita um recurso.
HEAD	solicita apenas o cabeçalho associado a um recurso.
PUT	envia dados a serem adicionados a um recurso.
POST	acrescenta um recurso.
DELETE	remove um recurso.

Tabela 4.2: *Status codes* definidos pelo protocolo HTTP

Código	Descrição
200	*OK*. Recurso retornado com sucesso.
302	*Redirect*. O recurso se encontra em outra localização.
404	*Not found*. O recurso não pôde ser encontrado.

4.1.3 Conexão

O protocolo HTTP é um protocolo de aplicação e, portanto, depende dos protocolos TCP e IP para que possa efetivamente trocar mensagens com os dispositivos espalhados pela *Internet*. O protocolo TCP foi escolhido por apresentar as seguintes características:

1. Transporte de dados livre de erros.
2. Entrega ordenada, isto é, os dados chegam à aplicação na mesma ordem em que foram enviados.
3. *Full-duplex*, isto é, permite a transferência simultânea de *stream* de dados (*bytes*) nas duas direções da comunicação cliente/servidor.

Sendo assim, para que duas aplicações HTTP possam se comunicar, é preciso que elas estabeleçam uma conexão TCP. Para estabelecer uma conexão com um servidor, é preciso antes conhecer seu endereço IP e o número da porta da aplicação com a qual deseja-se conectar. Servidores *Web* assumem a porta de número 80 como padrão para o protocolo HTTP e o endereço IP pode ser obtido através do DNS (*Domain Name System*). O DNS é um banco de dados distribuído que pode ser utilizado por aplicações TCP/IP para estabelecer um mapeamento entre *hostnames* (presentes em uma URL) e endereços IP como definido em [35].

A Figura 4.4 descreve o processo de conexão e visualização de uma página HTML por meio de um navegador *Web*:

1. O usuário digita a URL "ftp.arl.mil/mike/ping.html" no navegador.
2. O navegador identifica qual parte da URL refere-se ao *hostname*.
3. O navegador utiliza o DNS para traduzir o *hostname* em um endereço IP.
4. O navegador verifica se a URL contém um número de porta (exemplo: www.website.com**:8080**). Caso não haja, ele assume a porta de número 80.
5. O navegador estabelece uma conexão TCP com o servidor *Web* correspondente.
6. Uma vez estabelecida a conexão, o navegador envia uma requisição HTTP ao servidor.
7. O servidor envia uma resposta HTTP ao navegador.
8. A conexão é encerrada.
9. O navegador exibe a página recebida.

Apesar de aparentar um único recurso, deve-se perceber que uma página HTML pode ser na verdade um conjunto de recursos. Atualmente, a maioria das páginas possuem diversos recursos associados (arquivos Javascript, CSS, imagens, etc) que podem ou não estar espalhados por diferentes servidores. Nesse caso, os navegadores executam uma transação para recuperar o recurso base (aquele que referencia outros recursos) e, em seguida, realizam uma série de transações para obter os recursos adicionais como ilustra a Figura 4.5.

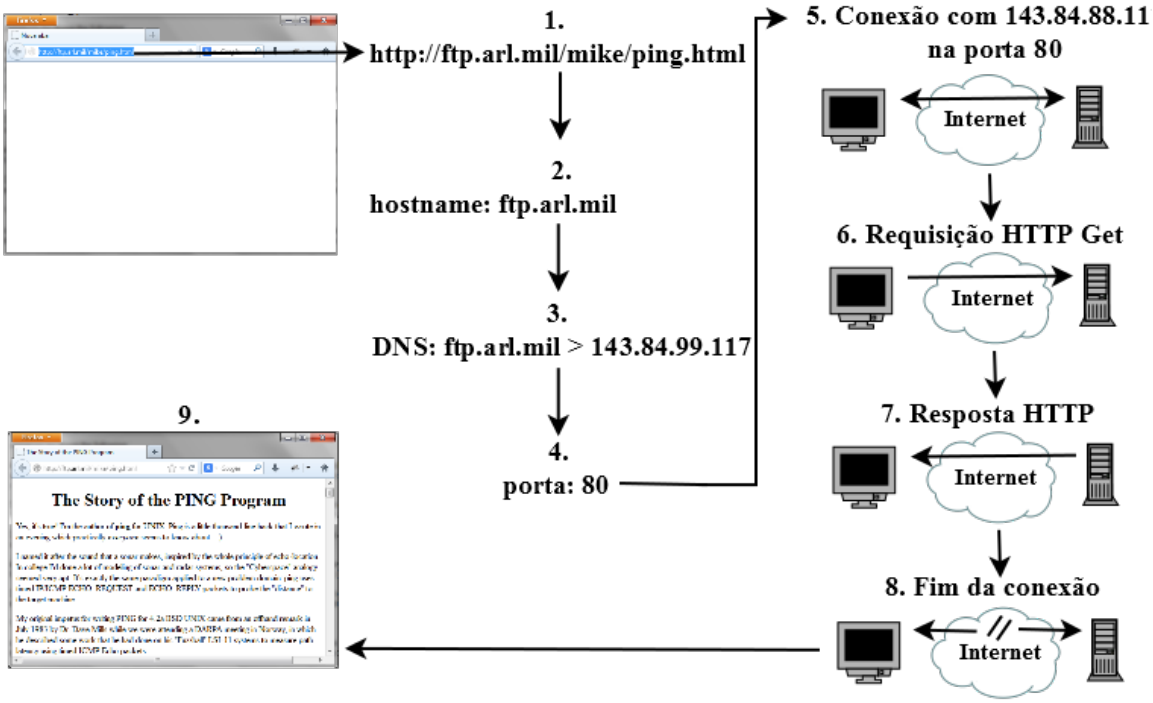


Figura 4.4: Conexão e visualização de uma página HTML - adaptada de [39]

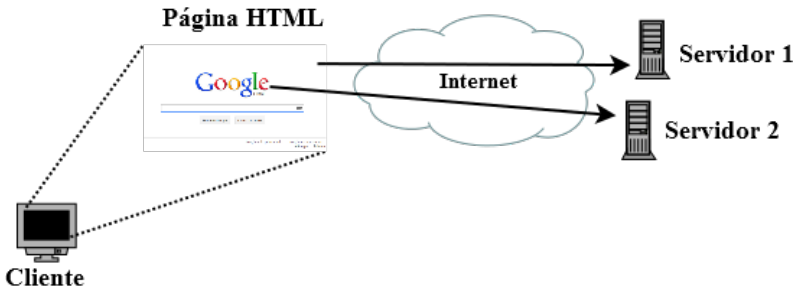


Figura 4.5: Recursos que compõem uma página HTML - adaptada de [39]

4.1.4 Servidores *Proxy*

Um servidor *proxy*, ou simplesmente *proxy*, é um servidor *Web* cuja função é intermediar as requisições e respostas de um cliente durante sua conexão com outro servidor *Web*. Dessa forma, as requisições vão do cliente para o *proxy* e do *proxy* para o servidor de destino final. As respostas, por sua vez, seguem o caminho inverso. É necessário destacar que os *proxies* têm acesso total às requisições e respostas do cliente, podendo alterá-las de acordo com seus próprios objetivos. Por isso, recomenda-se utilizar apenas os servidores que possam ser considerados confiáveis. Servidores *proxy* possuem diversas utilidades, dentre elas destacam-se:

1. Navegação "anônima": como discutido na seção anterior, ao acessar uma página da *Web*, é preciso estabelecer uma conexão TCP. Essa conexão fica registrada no servidor *Web* que se está acessando e esse registro pode ser utilizado para identificar quem e de onde essa página foi requisitada.

   O mesmo acontece quando se tenta acessar uma página através de um servidor *proxy*, porém, nesse caso, os dados registrados serão os do *proxy* e não mais os do cliente. Muitos *hackers* utilizam essa técnica (cascateando um grande número de *proxies*) para diminuir o risco de serem rastreados ao navegar na *Web*. Vale notar que ainda sim é possível rastreá-los, basta ter tempo e recursos suficientes para analisar os registros de todos servidores utilizados.

2. Restrição de conteúdo: é fácil perceber como os servidores *proxy* podem ser utilizados para filtrar ou registringir conteúdo, basta definir uma lista de URLs ou *hostnames* cujo acesso se quer proibir. Dessa forma, quando uma requisição com um destino que se encontra na lista for recebida, retorna-se uma página de erro ou até mesmo uma resposta com código 403 (*forbidden* - recurso proibido).

3. Aumento de desempenho: servidores *proxy* podem realizar o *caching* de determinadas transações HTTP (isto é, salvar as respostas das requisições) para melhorar o desempenho de acesso à *Internet*. Dessa maneira, ao receber uma requisição, o *proxy* verifica se possui uma resposta correspondente em seu *cache*. Se possuir, ele retorna a resposta armazenada (que é mais rápido que consultar o servidor de destino se o *proxy* estiver "perto" do usuário que solicitou o recurso). Caso contrário, ele realiza uma conexão com o servidor de destino e decide se deve ou não armazenar essa resposta no *cache*.

4.2 Filtragem de mensagens HTTP

Uma vez obtido o tráfego de rede de um determinado *host*, como demonstrado no capítulo 3.4, é necessário filtrar o tráfego HTTP. Analisando o tráfego a nível de pacotes, a filtragem pode ser realizada selecionando apenas aqueles nos quais verificam-se as seguintes condições:

1. IP de origem ou de destino é o IP do *host* monitorado.

2. Contém dados relativos ao protocolo TCP.

3. Número da porta de origem ou de destino TCP é 80.

Utilizando uma ferramenta de captura de pacotes como a *libpcap* [4] é possível configurar um filtro que realiza as verificações listadas. No entanto, é preciso lembrar que, tipicamente, ferramentas de captura de pacotes escutam tanto os pacotes que chegam a uma interface quanto os pacotes que saem dela.

Como o dispositivo que realiza o monitoramento (por meio do ataque *man-in-the-middle*) utiliza a mesma interface para receber e encaminhar o tráfego do roteador ao *host* monitorado e vice-versa, a ferramenta de captura irá exibir pacotes duplicados. Para remover as duplicatas, basta filtrar os pacotes pelo endereço MAC do *host* que realiza o monitoramento.

Essa abordagem de filtragem, apesar de simples, acaba se tornando um tanto trabalhosa. Um pacote possui seu tamanho limitado pelo MTU da rede que para o padrão Ethernet é 1500 bytes. As mensagens HTTP são na prática mensagens de texto e excedem com facilidade esse limite. Isso ocasiona a segmentação de pacotes TCP que deve ser lidada pelo programa caso ele trabalhe a nível de pacotes. Dessa forma, é preciso coletar e juntar todos os segmentos para então extrair as mensagens HTTP enviadas.

Após a filtragem, é preciso ainda distinguir as requisições das respostas. Isso pode ser feito verificando o IP de destino de cada pacote. Se o IP for igual ao do *host* monitorado, sabe-se que esse pacote contém uma resposta HTTP, caso contrário, contém uma requisição.

4.3 Visualização de páginas HTML

Além da filtragem, este trabalho propõe uma abordagem de visualização dos dados capturados em tempo real, que, até onde o autor saiba, é inovadora. À medida que páginas da *Web* forem requisitadas pelo *host* monitorado, as páginas retornadas são exibidas no navegador da máquina que realiza o monitoramento.

Para isso, antes da filtragem inicia-se um servidor *proxy* local. Durante a filtragem, armazena-se as transações HTTP interceptadas em um *cache* que pode ser acessado por esse *proxy*. O próximo passo consiste em identificar quais requisições buscam recuperar páginas HTML. Essa tarefa é um pouco mais complicada, pois não é possível extrair essa informação analisando apenas a requisição HTTP. A abordagem adotada, neste trabalho, consiste em analisar a resposta de cada transação de acordo com os seguintes passos:

1. Se o cabeçalho da resposta possuir o valor "text/html" para o campo "Content-type", sabe-se que aquela resposta possui conteúdo HTML (porém não garante ainda que contém uma página HTML).

2. Se a resposta estiver codificada em *chunks*, mecanismo introduzido pela versão 1.1 do protocolo HTTP, decodifica-se a mensagem como definido em [19].

3. Se a mensagem decodificada estiver comprimida (páginas HTML tipicamente são comprimidas no formato *gzip*), os dados são descompactados.

4. Por fim, realiza-se uma busca textual sobre os dados descompactados pelas sequências de caracteres "<html" e "</html>". Se a busca retornar resultado positivo, então considera-se que essa resposta contém uma página HTML.

Ao identificar uma resposta que contém uma página HTML, recupera-se a requisição associada e reconstrói-se a URL que a originou. O último passo consiste em acessar essa URL utilizando um navegador local configurado para acessar o *proxy* que foi iniciado anteriormente.

Isso pode ser feito utilizando ferramentas de automação de navegadores *Web*, que permitem controlar a grande maioria dos navegadores. Dessa forma, o *proxy* recebe a requisição, localiza em seu *cache* uma resposta correspondente à URL requisitada e retorna os dados capturados ao navegador local, permitindo assim a visualização do tráfego capturado.

4.4 Melhorando a filtragem e visualização

A visualização correta das páginas *Web* está diretamente relacionada aos dados filtrados que, por sua vez, correspondem aos dados que são enviados/recebidos pelo *host* monitorado.

Porém, muitas vezes, nem todos os dados necessários para a visualização são transmitidos, pois o protocolo HTTP prevê diversos mecanismos de *caching* de recursos, como detalhado em [39]. Essa prática normalmente está associada aos servidores *proxy*, porém é realizada também pela maioria dos navegadores atuais da seguinte forma:

1. Ao receber uma resposta do servidor, o navegador a armazena de acordo com informações contidas em cabeçalhos especiais que configuram como deve ser feito o *caching*.
2. Quando o navegador realiza uma requisição ele primeiro verifica se a versão que tem armazenada é válida (recente o suficiente). Se for, ele responde a requisição com os dados armazenados sem transmití-la ao servidor pretendido.
3. Caso o navegador não tenha condições de determinar com certeza se sua versão é válida, ele transmite a requisição ao servidor incluindo cabeçalhos que caracterizam a versão que está no *cache*.
4. O servidor, por sua vez, realiza suas próprias verificações analisando os cabeçalhos enviados na requisição e retransmite a resposta se conseguir determinar que a cópia armazenada do recurso não é válida ou caso não consiga determinar sua validade.

Os cabeçalhos que tornam possível o *caching* podem ser agrupados em cabeçalhos que são enviados em mensagens de requisição e aqueles enviados em mensagens de resposta.

- Cabeçalhos de requisição:
  - *If-Modified-Since*: especifica uma data que será utilizada para verificar se a versão de um recurso é a mais recente. Se existir, no servidor, uma versão com data de modificação mais recente que a data especificada, sabe-se que o recurso armazenado está desatualizado.
  - *If-None-Match*: permite verificar se um recurso é recente de acordo com *tags* especiais definidas pelo servidor. A resposta do servidor contém uma *tag* que fica armazenada no *cache*. Dessa maneira, ao realizar uma nova requisição o valor da *tag* é recuperado e enviado como valor deste cabeçalho.

Tabela 4.3: Possíveis valores do cabeçalho *Cache-Control*

Valor	Descrição
no-store	indica que uma cópia da resposta não dever ser armazenada.
no-cache	indica ao gerenciador do *cache* que ele não deve servir uma cópia desta resposta sem antes verificar com o servidor de origem se existe uma versão mais recente. (este cabeçalho com esse valor pode ser usado também em requisições).
max-age	define um período como um número de segundos a partir da data de envio da resposta dentro do qual o documento ainda será considerado válido.

- Cabeçalhos de resposta:
  - *Expires*: define uma data a partir da qual aquela resposta estará desatualizada e portanto não pode mais ser servida do *cache*.
  - *Cache-control*: especifica a política de *caching* determinada pelo servidor. A Tabela 4.3 apresenta os possíveis valores.
  - *Pragma*: este cabeçalho é definido no protocolo HTTP/1.0 e é utilizado nas versões mais novas para permitir *backwards compatibility*. Normalmente tem o valor *no-cache* cuja função é a mesma do cabeçalho *Cache-control: no-cache*.

Apesar de melhorar o tempo de resposta de páginas *Web* tornando a interação com usuário mais fluida, esse esquema constitui um obstáculo para a realização do monitoramento proposto por esse trabalho, pois arquivos presentes no *cache* do *host* monitorado provavelmente não estarão no *host* que monitora.

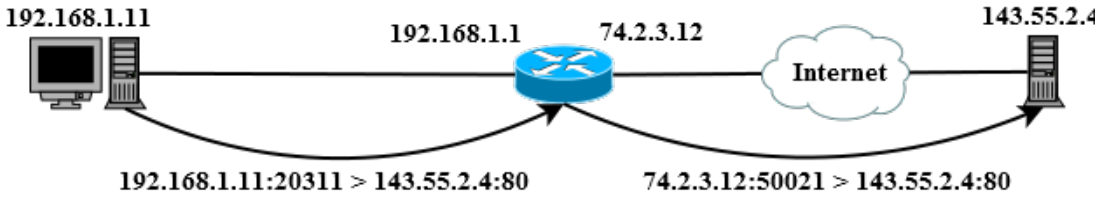
Levando isso em consideração, percebeu-se que seria preciso alterar as requisições e respostas enviadas/recebidas pela máquina monitorada para contornar esse problema, da seguinte maneira:

1. Remover todos os cabeçalhos *If-Modified-Since* e *If-None-Match* presentes nas requisições enviadas.
2. Adicionar/alterar os cabeçalhos *Cache-Control* e *Pragma* com os valores *no-store* e *no-cache* respectivamente.
3. Adicionar/alterar o cabeçalho *Expires* configurando a data de expiração para uma data antiga: "*Fri, 01 Jan 1990 00:00:00 GMT*".

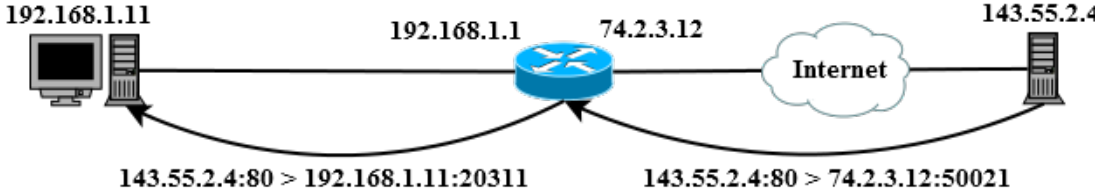
Para isso, foi preciso mudar a abordagem de filtragem, tendo em vista que apesar de simples ela torna muito difícil a realização das manipulações necessárias.

A solução encontrada foi adicionar outro servidor *proxy* com o objetivo de intermediar todas as conexões do navegador da máquina sendo monitorada. Para tal seria preciso redirecionar o tráfego da máquina alvo para o *proxy* local da máquina que a monitora.

No entanto, a ferramenta utilizada até então para analisar/filtrar o tráfego era uma ferramenta de captura de pacotes apenas, e esse tipo de ferramenta normalmente apenas copia os dados dos pacotes sem alterar seu conteúdo nem sua direção.



(a) Alteração de um pacote com destino a um *host* externo



(b) Alteração de um pacote com destino ao roteador

Figura 4.6: Etapas do processo de NAT

Portanto foi preciso encontrar uma ferramenta que possibilitasse esse tipo de manipulação e elaborar um esquema de redirecionamento. O esquema elaborado teve como inspiração o princípio de funcionamento do NAT (*Network Address Translation*) [18].

O NAT é um processo comumente implementado por roteadores cujo objetivo é permitir que diferentes máquinas de uma mesma sub-rede conectem-se à internet utilizando um único endereço IP. O processo consiste em alterar os dados de pacotes TCP/IP que trafegam em duas direções como mostram as Figuras 4.6a e 4.6b:

1. De um *host* interno a um *host* externo: como todos pacotes com destino fora de uma sub-rede são encaminhados ao roteador, ao receber um pacote desse tipo, o roteador altera o endereço de IP e o número da porta de origem, trocando essas duas informações pelo seu próprio endereço e uma porta arbitrária.

   Em seguida, ele armazena, em uma tabela, o IP de origem e número de porta originais, associando esses dados ao número de porta escolhido e o IP de destino. Por fim ele encaminha o pacote ao destinatário pretendido.

2. De um *host* externo ao roteador: ao receber um pacote de um *host* externo, o roteador verifica se ele deve ser encaminhado a um *host* interno consultando o endereço IP de origem do pacote e o número de porta de destino em sua tabela.

   Dessa maneira, se existir uma entrada correspondente na tabela ele sabe que o pacote recebido deve ser encaminhado. Nesse caso, ele desfaz as alterações feitas na primeira etapa, isto é troca o IP e número de porta de origem por aqueles presentes na tabela.

A técnica implementada consiste, primeiramente, em filtrar todos os pacotes que tenham número de porta de destino 80 e IP de destino fora da sub-rede local. Esses pacotes têm o IP de destino trocados para o IP da máquina que realiza o monitoramento e o número de porta de destino trocada pelo número da porta na qual o servidor *proxy* foi configurado.

Assim como no NAT, os dados alterados foram colocados em uma tabela para permitir a alteração dos pacotes que seguem o caminho inverso. Esse processo por sua vez consiste

em trocar o número da porta de origem para 80 e o IP de origem para o IP externo à sub-rede local.

Dessa maneira, o *proxy* passou a ter controle sobre todas as requisições que são enviadas e todas as respostas recebidas pelo *host* alvo. Assim tornou-se possível desabilitar as configurações de *cache* indesejadas, melhorando a visualização dos dados interceptados pelo navegador *Web* da *host* que realiza o monitoramento.

4.5 Protocolo HTTPS

Como mostrado até esse ponto, o protocolo HTTP está vulnerável a ataques do tipo *man-in-the-middle*, o que permite capturar, visualizar e alterar todos os dados transmitidos em transações HTTP.

O objetivo do protocolo HTTPS (*Hypertext Transfer Protocol Secure*) é tornar o protocolo HTTP seguro, de maneira que ele possa ser utilizado para realizar transações importantes como operações bancárias ou armazenamento de arquivos confidenciais.

Para alcançar esse objetivo, ele prevê a adicão de uma camada de transporte segura acima do protocolo TCP, encapsulando assim o protocolo HTTP. Essa camada faz uso de técnicas de criptografia que buscam garantir que as mensagens enviadas são abertas apenas pelos destinatários pretendidos, bem como prover mecanismos para comprovar a autenticidade de servidores.

Essas técnicas são definidas pelo protocolos SSL/TLS (*Secure Sockets Layer/Transport Layer Security*). Apesar de não ser o foco deste trabalho, um estudo geral sobre criptografia e SSL foi realizado, conforme apresentado no apêndice A.

A idéia central do protocolo HTTPS é simples: primeiramente, cliente e servidor trocam mensagens para estabelecer parâmetros do protocolo SSL. Uma vez estabelecida uma conexão SSL (comumente chamada de conexão segura), ambas as partes podem enviar mensagens HTTP normalmente.

O uso do protocolo HTTPS é opcional e é determinado pelo prefixo da URL de um recurso. Como explicado na seção 4.1.3, para transmitir mensagens HTTP é preciso primeiro estabelecer uma conexão TCP utilizando a porta 80 como padrão. De forma análoga, o protocolo HTTPS está associado à porta 443. Essa distinção de portas é feita para evitar que os servidores HTTP tentassem interpretar mensagens SSL como mensagens HTTP, ocasionando assim um comportamento errôneo.

Uma prática muito utilizada que vem perdendo sua popularidade consiste em utilizar o protocolo HTTPS apenas para transações que envolvem recursos que exigem uma conexão segura. Esse modelo faz sentido, pois estabelecer uma conexão SSL é um processo muito mais oneroso que estabelecer uma conexão TCP comum.

No entanto, essa prática aliada à ampla utilização do protocolo HTTP acabam contribuindo para a exposição dos servidores *Web* e seus usuários à técnica conhecida como SSL *Stripping* [25, 13]. Para entender essa técnica, primeiro é preciso entender o comportamento dos servidores *Web* quando um cliente solicita, através de uma conexão comum, um recurso que exige uma conexão segura:

1. O cliente conecta-se ao servidor na porta 80 e envia uma requisição HTTP ao servidor.

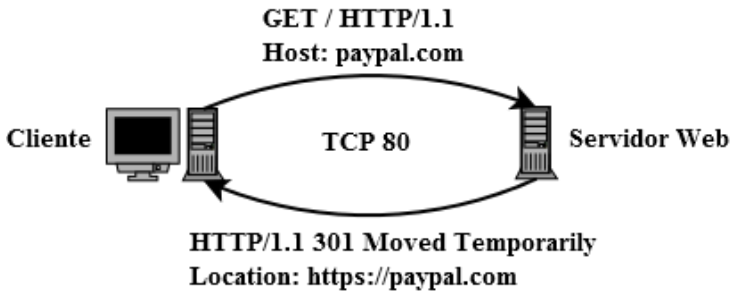


Figura 4.7: Redirecionamento da requisição de um recurso que exige uma conexão segura

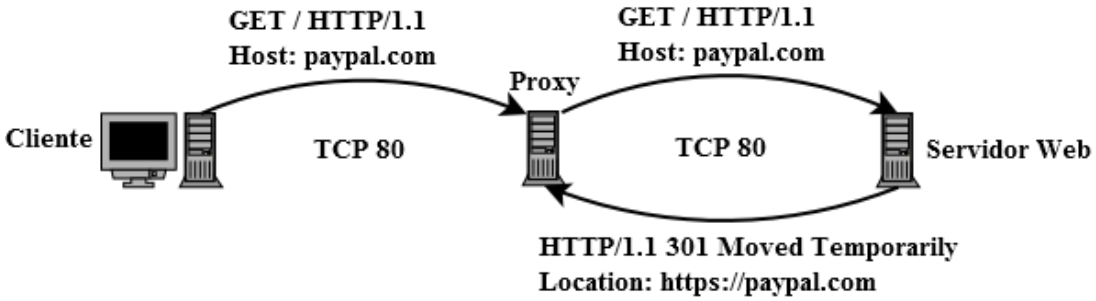
2. O servidor verifica que o recurso solicitado só pode ser transmitido por meio de uma conexão segura. Dessa forma, ele envia uma mensagem de redirecionamento (código de *status* 301 ou 302) adicionando a URL que o cliente deverá acessar para obter o recurso no cabeçalho denominado *Location*.
3. O cliente recebe a resposta, analisa a URL presente no cabeçalho *Location* e com base nela inicia uma conexão SSL.

A Figura 4.7 ilustra o processo, enquanto a Figura 4.8 apresenta como esse comportamento pode ser reconhecido e explorado por um servidor *proxy* malicioso:

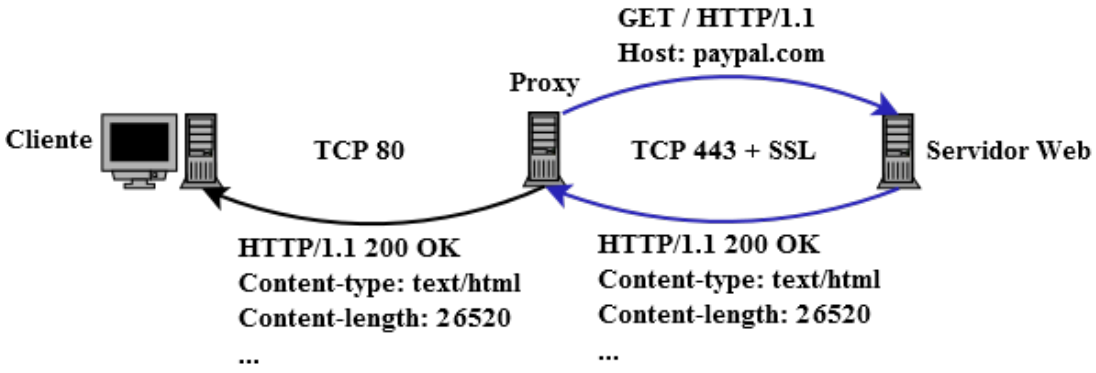
1. O cliente conecta-se ao *proxy* na porta 80 e envia uma requisição HTTP ao servidor.
2. A requisição chega ao *proxy*, que a encaminha para o servidor pretendido.
3. O servidor verifica que o recurso solicitado requer uma conexão segura e envia a mensagem de redirecionamento ao *proxy*.
4. O *proxy* detecta que recebeu uma mensagem de redirecionamento para uma URL cujo protocolo é HTTPS. Em seguida, ele estabelece uma conexão SSL legítima com o servidor e reenvia a requisição HTTP.
5. O servidor agora recebe uma requisição por meio de uma conexão SSL e envia uma resposta com o recurso solicitado.
6. O *proxy* repassa a resposta com o recurso ao cliente.
7. O cliente recebe a resposta.

É importante notar que o *proxy* somente deve estabelecer a conexão SSL quando os parâmetros *host* e URI da URL da requisição e do redirecionamento do servidor forem iguais, caso contrário o *proxy* deve repassar a mensagem de direcionamento. Isso deve ser feito para que o cliente exiba a URL mais próxima da URL relacionada ao recurso sendo retornado.

Por exemplo: a requisição para a URL *http://mail.google.com* gera um redirecionamento para a URL *https://accounts.google.com*. Supondo que o cliente nessa situação seja um navegador, caso o redirecionamento não seja transmitido, navegador irá exibir o conteúdo relacionado à segunda URL, porém a URL que constará na barra de endereços será a primeira, o que pode levar um usuário perceptivo a desconfiar que algo está errado.



(a) Envio da requisição original do cliente e obtenção de uma resposta de redirecionamento



(b) Estabelecimento de uma conexão SSL e repasse da resposta obtida ao cliente

Figura 4.8: Redirecionamento intermediado por um servidor *proxy* malicioso

Conforme dito anteriormente, esta técnica está centrada no fato de que a navegação de um usuário na *Internet* normalmente começa com páginas HTTP, até mesmo porque a maioria dos usuários escrevem "site.com" na barra de endereços, ao invés de "http://site.com" ou "https://site.com".

Além disso, para que ela seja bem sucedida, é preciso também alterar o conteúdo das páginas HTML servidas pelo *proxy*, pois elas podem conter *links* com URLs de destino que possuem o protocolo HTTPS especificado.

Por fim, caso a página servida tenha sido originada de uma conexão SSL, é preciso verificar as URLs de todos os recursos referenciados por ela, pois elas podem ser relativas à URL do recurso pricipal. Como a URL do recurso principal se encontra alterada do ponto de vista do cliente, esses recursos somente serão carregados corretamente caso suas URLs sejam transformadas de relativas em absolutas levando em consideração a URL original (cujo prefixo é *https://*). A Tabela 4.4 apresenta como devem ser tratadas URLs relativas relacionadas ao recurso *https://paypal.com/main*.

O SSL *Stripping* consegue burlar o propósito do protocolo HTTPS de uma maneira simples e elegante. Apesar de existirem outras técnicas que cumprem o mesmo objetivo, o SSL *Stripping* é uma das técnicas que possui maior índice de sucesso, conforme mostrado em [13]. No entanto, como todas as outras técnicas, possui um conjunto de limitações:

1. Um usuário perspicaz pode perceber que o seu navegador não apresenta os sinais de que foi estabelecida uma conexão segura ao acessar um site que frequenta periodicamente, percebendo, assim, que está sendo vítima de um ataque.

Tabela 4.4: Resultado do tratamento de URLs relativas relacionadas ao recurso *https://paypal.com/main*

URL	Resultado
//paypal2.com/main	*https://paypal2.com/main*
/menu/options	*https://paypal.com/menu/options*
video.wmv	*https://paypal.com/video.wmv*

2. A quantidade de pessoas que digitam o endereço dos sites vem diminuindo pois os navegadores buscam cada vez mais facilitar o acesso e configuração de atalhos para abrir as URLs mais frequentadas.

3. Alguns navegadores *Web* estão programados para preferir a versão que especifica o protocolo HTTPS de URLs presentes no histórico. Dessa forma, se a URL *https://site.com* estiver presente no histórico, quando o usuário tentar acessar *site.com* o navegador automaticamente assume o protocolo HTTPS não o HTTP.

4. Diversas formas de detecção e prevenção do SSL *Stripping* já foram elaboradas como descrito em [30, 27, 44].

Capítulo 5

Desenvolvimento do *Webspy*

O *Webspy* foi concebido e desenvolvido utilizando as linguagens de programação C e C# com foco voltado exclusivamente para o Sistema Operacional *Windows*. Ele foi testado em sua maior parte nas versões *Windows* 7 e 8. O modelo de interface com o usuário adotado foi o de linha de comando devido à simplicidade de implementação.

Desde o princípio, sabia-se que o *Webspy* iria precisar de diversas ferramentas para cumprir seus objetivos e que nem todas estariam disponíveis em uma linguagem de programação apenas. Por isso foi adotado um modelo de desenvolvimento em módulos composto de 5 etapas:

1. Divisão das funcionalidades propostas em módulos.
2. Implementação de cada módulo utilizando a linguagem mais conveniente.
3. Realização de testes em cada um dos módulos separadamente.
4. Integração dos módulos.
5. Realização de testes de integração.

Dessa forma, foram definidos 4 módulos:

1. Módulo de Varredura.
2. Módulo de ARP *Spoofing*.
3. Módulo de Visualização de páginas HTML.
4. Módulo de *Playback*.

As Seções de 5.1 a 5.4 apresentam as funcionalidades de cada módulo, bem como detalhes e decisões de implementação, enquanto que a Seção 5.5 discute a integração entre os módulos e os resultados obtidos.

5.1 Módulo de Varredura

O módulo de varredura tem o objetivo de exibir ao usuário todos os dispositivos que se encontram em uma rede, utilizando as técnicas de Ping *sweep* e *ARP* sweep descritas no capítulo 3. Esse módulo foi desenvolvido com a linguagem C utilizando as bibliotecas *WinPcap* [9] e *libnet* [5] para captura e injeção de pacotes respectivamente.

5.1.1 Desenvolvimento

A inspiração inicial veio do programa Ping, que consiste em enviar uma requisição ICMP, escutar um determinado tempo por uma uma resposta e repetir o processo 4 vezes.

O primeiro protótipo desenvolvido implementava o Ping *sweep*. Ele identificava primeiramente seu próprio endereço IP e, assumindo que se encontra em uma rede de classe C, iniciava o processo de envio de requisições para endereços IP com *hostid* de 1 a 254.

Esse processo consistia em enviar uma requisição ICMP e esperar 3 segundos por uma resposta. Caso a resposta fosse detectada, o endereço IP de origem era exibido o IP era marcado como conectado.

Essa abordagem, apesar de eficaz, era muito ineficiente, tendo em vista que era preciso esperar por um determinado tempo uma resposta e só então enviar uma requisição para o próximo IP.

Dessa forma, na segunda versão do protótipo, verificou-se que era possível melhorar o desempenho fazendo com que o programa tivesse duas *threads*: uma *thread* que injetava pacotes ICMP para todos os possíveis *hosts* da rede e outra *thread* que apenas escutava as respostas. Assim foi possível tornar a varredura muito mais eficiente.

Inicialmente pensou-se em exibir apenas os endereços IP e MAC dos *hosts* descobertos, porém percebeu-se que essa informação somente não era de muita ajuda, pois nem sempre é conhecido o IP do dispositivo que se quer monitorar.

Para resolver esse problema foi utilizada a API de *Sockets* do *Windows*, que permite recuperar um nome associado a um IP. Normalmente esse nome é o nome dado ao dispositivo durante a instalação de seu Sistema Operacional (que costuma seguir o padrão NomeDono-PC ou NomeDono-Note). No entanto, essa informação não está disponível para todos os dispositivos (nesse caso o endereço IP é exibido novamente).

O protótipo que implementa o ARP *Sweep* seguiu a mesma arquitetura do protótipo do Ping *Sweep*, porém utilizando mensagens do protocolo ARP.

5.1.2 Resultado

Ambos protótipos possuem apenas um parâmetro de entrada: o nome da interface na qual deve ser realizada a varredura. Caso esse parâmetro não seja informado (ou se não for encontrada nenhuma interface com nome correspondente) ele exibe um menu que lista as interfaces encontradas, permitindo assim que o usuário escolha uma, como mostra a Figura 5.1. As Figuras 5.2a e 5.2b apresentam a saída da última versão dos prótipos implementados.

5.2 Módulo de ARP *Spoofing*

O módulo de Arp *Spoofing* tem o objetivo de implementar a técnica descrita na Seção 3.4 de forma que o tráfego de rede um determinado *host* de uma rede possa ser interceptado. Ele foi desenvolvido utilizando as mesmas ferramentas de captura e injeção de pacotes utilizadas no módulo de varredura.

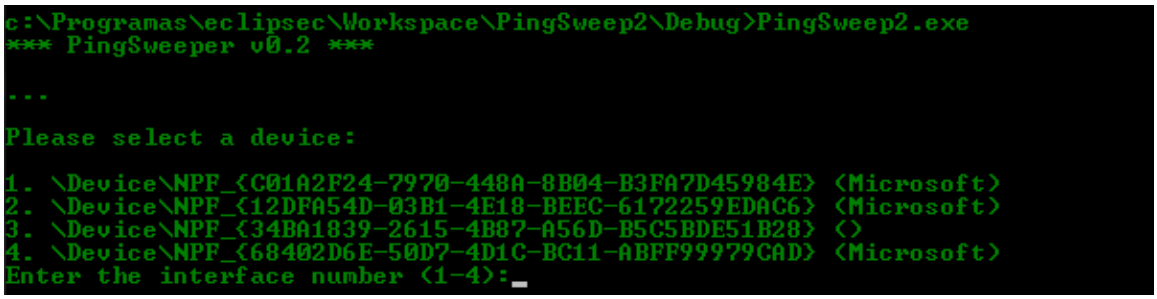
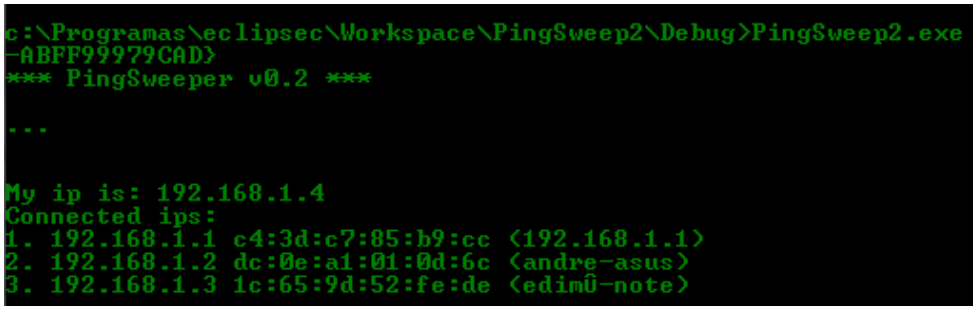
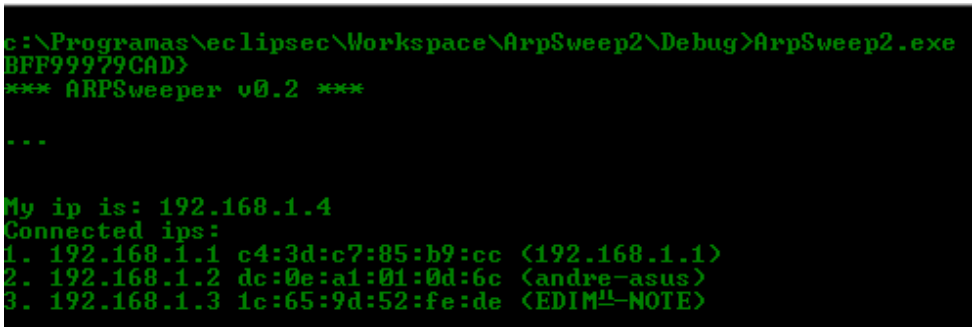


Figura 5.1: Menu de seleção de interface



(a) Saída do Ping *Sweeper*



(b) Saída do ARP *Sweeper*

Figura 5.2: Saídas dos protótipos que implementam ARP e Ping *sweep*

5.2.1 Desenvolvimento

O protótipo inicial consistia em uma aplicação com duas *threads*: a primeira era encarregada de fazer o ARP *Poison* com um intervalo fixo de 1 segundo, e a segunda, de escutar os pacotes e realizar o encaminhamento (*Relay*).

Os testes deste módulo consistiam em conectar duas máquinas a um roteador por meio de cabos. Em seguida iniciava-se o protótipo em uma das máquinas enquanto, na outra, abria-se um navegador *Web*. Por fim, verificava-se o tempo de resposta para que o navegador visualizasse determinadas páginas da *Web*.

O desempenho do primeiro protótipo foi muito baixo, e, durante os testes, percebeu-se que a velocidade da conexão da vítima caía drasticamente: a visualização de uma página simples como a do *Google* demorava cerca de 10 segundos e páginas mais complexas apresentavam um *timeout* de conexão.

Dessa maneira, decidiu-se fazer uma segunda versão do protótipo, na qual cada *thread* foi separada em um único processo, porém o desempenho nao melhorou. O último protótipo criado adotou outra abordagem: ele concetrava-se em realizar apenas o ARP *Poison* e, para realizar o *Relay*, foi configurada a opção *IPEnableRouter* do *Windows*.

Essa configuração permite que o *Windows* roteie pacotes recebidos por ele cujo IP de destino não é o seu. Esta versão apresentou desempenho satisfatório de maneira que não era perceptível nenhuma alteração drástica na conexão da vítima.

Esse resultado levou a acreditar que havia algo errado na implementação do *Relay* das versões anteriores ou então que as ferramentas utilizadas tinham baixo desempenho.

Enquanto pensava-se sobre o assunto, decidiu-se experimentar ferramentas prontas de ARP *Spoofing* disponíveis na *Internet*. Algumas ferramentas foram testadas sem muito sucesso (várias delas dispararam o alerta de vírus do anti-vírus *Avira*) até que foi encontrada uma denominada APE (ARP *Poison Engine*) [42].

Essa ferramenta, além de ter o melhor desempenho observado, apresentava um sistema de *log* organizado, o que permitiu acompanhar todos os passos realizados pela ferramenta, como mostra a Figura 5.3. Dessa maneira verificou-se que o *Relay* era feito utilizando uma ferramenta de administração do *Windows* chamada *netsh*.

Esse ferramenta era utilizada para o configurar o uso do NDP (*Neighbor Discovery Protocol*) [37]. Esse protocolo está, na verdade, associado à versão 6 do protocolo IP (porém funciona também com a versão 4) e permite que *hosts* se comuniquem e troquem pacotes a nível de camada 2 (o que justifica o desempenho observado).

Mais tarde descobriu-se que o problema de desempenho dos primeiros protótipos estava associada a um parâmetro de *timeout* de leitura da biblioteca *WinPcap*. Uma vez reduzido o valor desse parâmetro ao mínimo, o desempenho equiparou-se à do protótipo que usava a opção *IPEnableRouter* para realizar o *Relay*.

5.2.2 Resultado

Todos os protótipos implementados recebem, como entrada, o nome da interface (parâmetro opcional) e o IP da vítima e, como saída, exibem o número de iterações do ARP *Poison* a cada segundo. O protótipo que faz *Relay* exibe também a direção dos pacotes encaminhados utilizando a letra R para representar o roteador e V, para a vítima como mostra a Figura 5.4. A versão final implementada adotou o modelo inicial que utiliza duas *threads*.

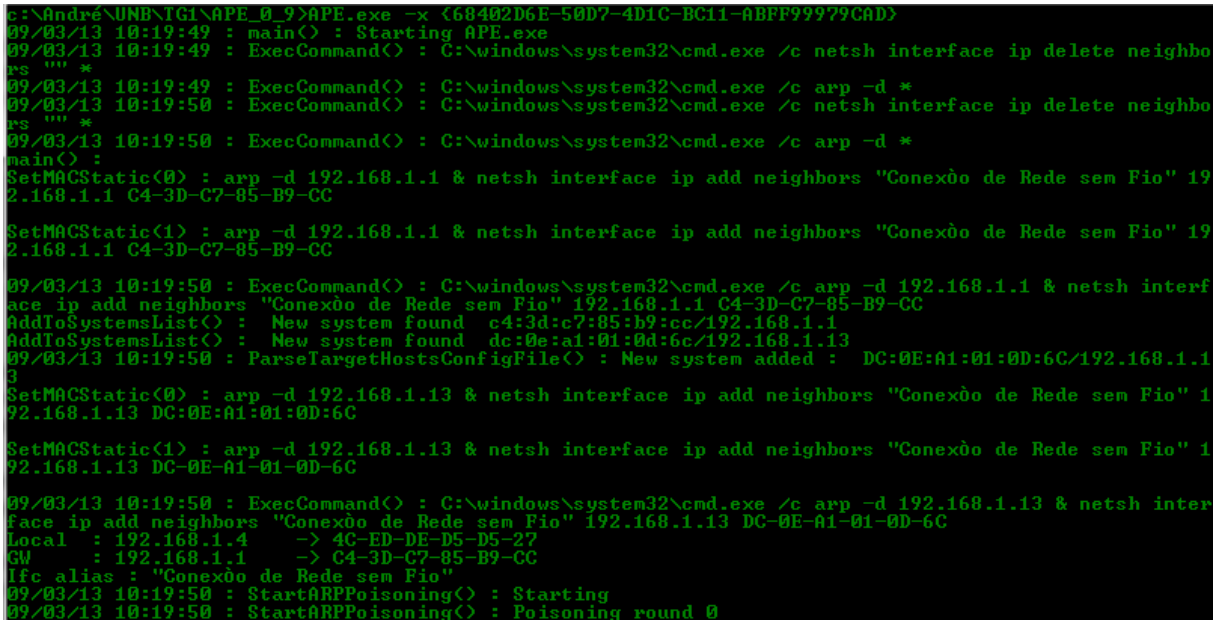


Figura 5.3: Ferramenta APE (ARP *Poison Engine*)

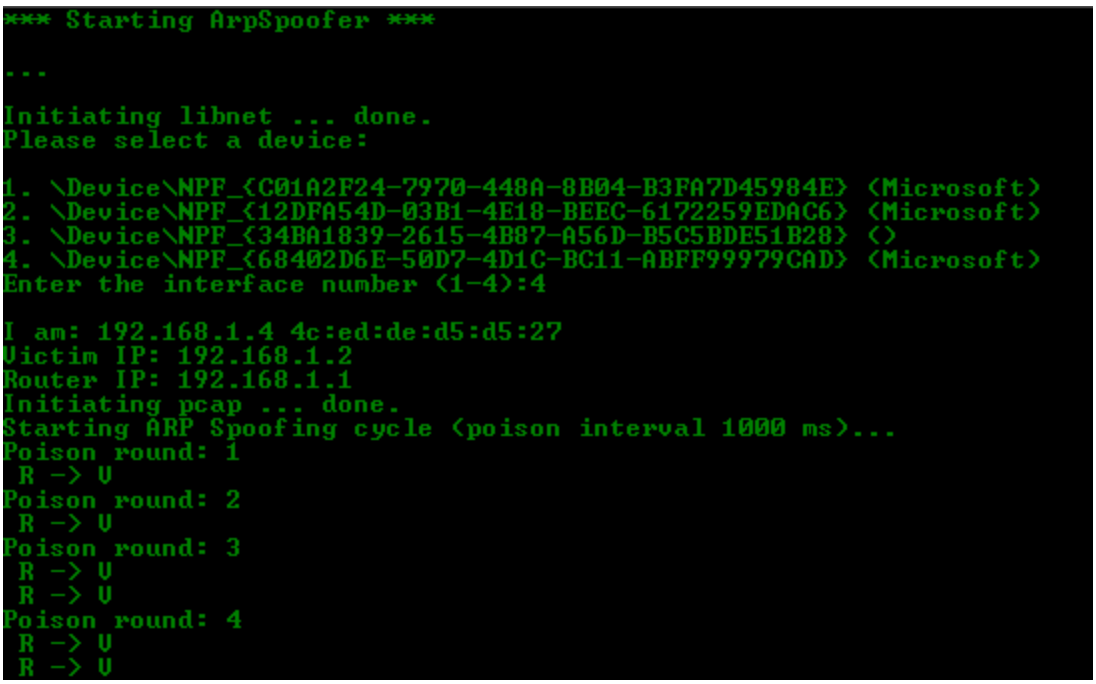


Figura 5.4: Prótipo implementado que realiza ARP *Poison* e *Relay*

5.3 Módulo de visualização de páginas HTML

O módulo de visualização tem o objetivo de permitir que o usuário possa visualizar, em seu navegador local, as páginas *Web* presentes no tráfego de rede interceptado. Esse módulo foi desenvolvido em C# e constituiu o maior desafio de implementação. Foram construídas duas versões que correspondem às duas abordagens de filtragem e visualização apresentadas no capítulo 4. A divisão foi necessária pois as versões exigiam arquiteturas diferentes.

5.3.1 Primeira versão

A primeira versão implementa a idéia mais simples de filtragem de tráfego *Web*. Apesar da idéia ser simples, a implementação foi muito trabalhosa por tratar a questão da filtragem a nível de pacotes. Essa versão utilizou as seguintes bibliotecas:

1. *PcapDotNet* [2]: *wrapper* para C# da biblioteca *WinPcap* que permitiu a captura dos pacotes interceptados.
2. *TrotiNet* [6]: permitiu implementar servidores *proxy* HTTP com simplicidade e flexibilidade.
3. *Selenium* [3]: permitiu controlar navegadores *Web* atuais.

Desenvolvimento

Inicialmente definiu-se uma arquitetura que permitesse a implementação da abordagem proposta, como mostra a Figura 5.5. O primeiro passo consistiu em inicializar um servidor *proxy* programado para responder requisições com respostas contidas em um *cache* local. Esse *proxy* foi implementado de maneira que, caso a resposta de uma requisição não estivesse presente no *cache* ele transmitiria a requisição ao servidor pretendido e, em seguida, retornaria a resposta obtida.

Uma vez configurado o *proxy* (*Browser Proxy*), os pacotes foram capturados e filtraram-se aqueles que possuíam: a porta de número 80 do protocolo TCP especificada, o endereço IP da vítima do *ARP Spoofing* e o endereço MAC de destino do *host* que faz o *Spoofing* (para remover duplicatas como discutido na seção 4.2).

Em seguida, os pacotes foram separados em fluxos, que representam uma conexão TCP e, portanto, são diferenciados por um endereço IP e um número de porta TCP. Um fluxo contém todos os pacotes relativos a uma transação HTTP, sendo que um fluxo pode ter diversas transações. Os fluxos e suas transações foram construídos filtrando-se primeiramente apenas os pacotes cujo *payload* TCP era maior que zero. Em seguida, seguiu-se o seguinte processo:

1. Se for detectado um pacote com o endereço IP de destino diferente do IP da vítima:
   (a) Converte-se os *bytes* do *payload* TCP em uma *string* e verifica-se se ela pode ser dividida em linhas.
   (b) Se puder, verifica-se se a primeira linha é uma linha de requisição HTTP válida.

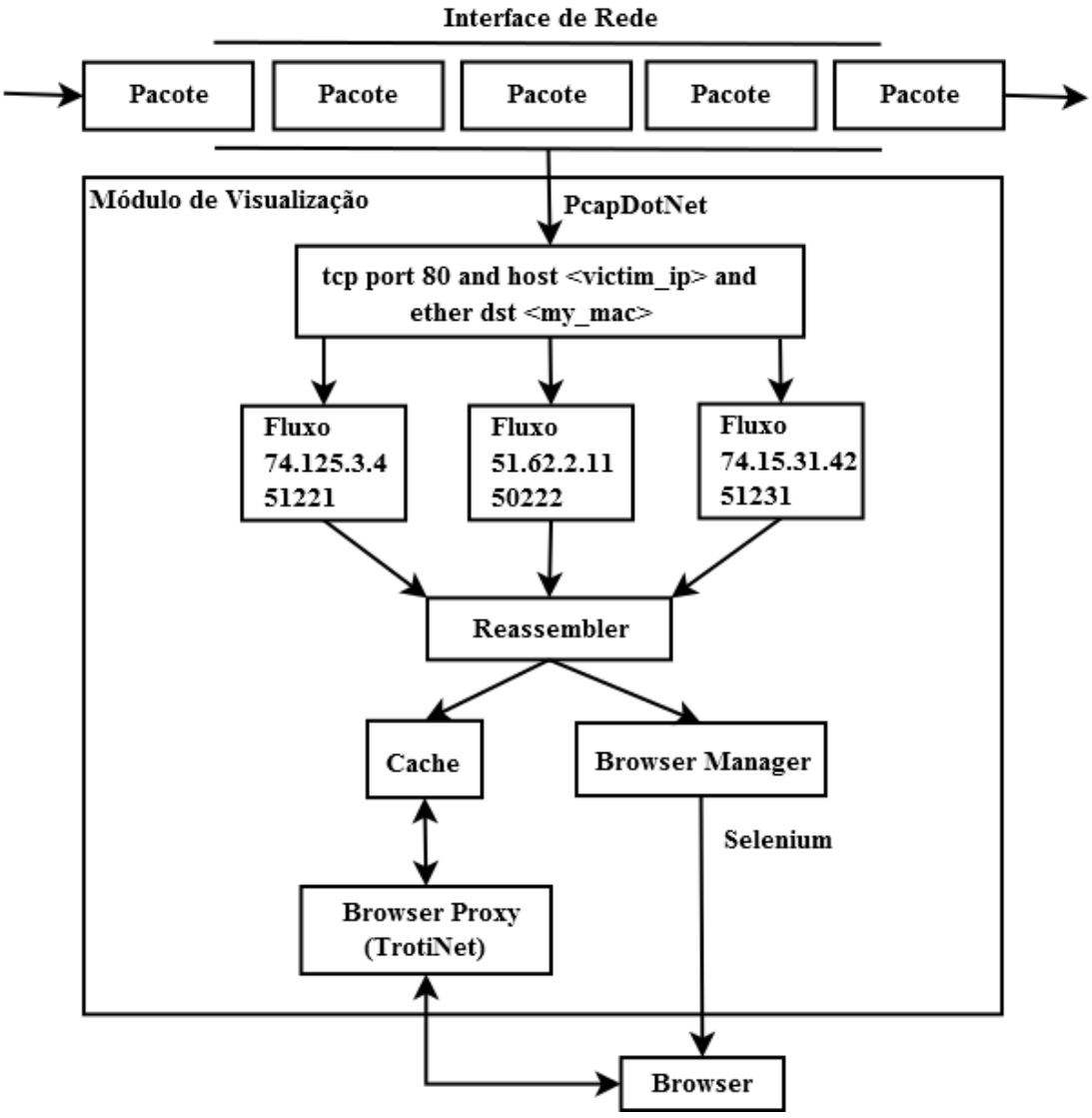


Figura 5.5: Arquitetura da implementação da primeira abordagem de filtragem

(c) Se for, verifica-se se já existe um fluxo com o endereço IP de destino e número de porta de origem correspondente aos do pacote.

(d) Se existir, cria-se uma nova transação adicionando a requisição HTTP encontrada.

(e) Se não existir, cria-se um novo fluxo e, em seguida, a nova transação com a requisição encontrada.

(f) Por fim, marca-se a transação adicionada como a transação atual para aquele fluxo.

2. Se for detectado um pacote com o endereço IP de destino igual ao da vítima:

(a) Procura-se um fluxo que tenha IP e número de porta iguais ao endereço IP de origem e número da porta de destino do pacote respectivamente.

(b) Se existir, adiciona-se o pacote à transação atual do fluxo.

(c) Se não existir fluxo ou se o fluxo encontrado não possuir uma transação marcada como atual, o pacote é descartado.

O próximo passo (implementado pelo componente chamado *Reassembler*) consistiu em analisar cada transação de cada um dos fluxos para verficar se foram coletados todos os pacotes que continham uma resposta HTTP. Esses pacotes continham segmentos TCP cuja ordem foi verificada para que fossem remontados. Isso foi feito porque a ferramenta de captura não garante que a ordem de entrega será mantida.

Uma vez remontada uma resposta HTTP, a transação é considerada completa e é enviada ao *cache* do servidor *proxy*. O *cache* foi implementado como uma tabela *hash* de transações, que são indexadas pela URL relativa à requisição da transação.

Além disso, a resposta HTTP é enviada também ao gerenciador do navegador *Web* (*Browser Manager*). Esse componente mantém uma tabela *hash* de instâncias de um navegador *Web* configuradas para utilizar o *Browser Proxy* e é indexada pelo *host* da URL sendo visualizada. Cada instância corresponde a uma janela do navegador.

Dessa forma, ao receber uma resposta HTTP, o *Browser Manager* a analisava para verificar se continha uma página HTML. Se essa condição fosse verificada, era recuperada a URL da requisição correspondente à resposta. Em seguida, o componente verificava se já existia uma instância do navegador *Web* escolhido associada ao *host* da URL recuperada.

Se existisse, a instãncia era recuperada, caso contrário era criada uma nova. Por fim, era transmitido um comando para que a instância acessasse a URL.

Foram testados dois navegadores: *Chrome* e *Firefox*. Foi escolhido o *Firefox*, pois para estabelecer uma conexão com o *Chrome* era preciso utilizar um aplicativo denominado *ChromeDriver* que apresentava um comportamento instável quando as janelas do navegador era fechadas pelo usuário. Por sua vez, o *Firefox* não necessitava de nenhum recurso adicional e mostrou-se mais estável.

Para que fosse possível desempenhar todas as funções descritas de maneira eficiente, foi preciso tornar o protótipo *multi-thread*. Sendo assim, foram definidas 3 *threads* principais:

1. *Thread* de escuta de pacotes: responsável por escutar os pacotes e construir os fluxos.

2. *Thread* de remontagem: responsável pela remontagem dos segmentos TCP das respostas, organização das transações HTTP, armazenamento de transações no *cache* e envio de respostas ao *Browser Manager*.

3. *Thread* do servidor *proxy*: responsável por realizar as funções de organização e manutenção do servidor *proxy* que, por sua vez, gera uma nova *thread* para cada requisição recebida.

Além disso, para melhorar o tempo de resposta do protótipo, o *Browser Manager* foi implementado de maneira a gerar uma nova thread a cada resposta HTTP recebida.

Os testes utilizaram a ferramenta APE (pois ela tinha o melhor desempenho observado até então) e foram conduzidos da mesma maneira que no módulo de ARP *Spoofing*. Testes inciais revelaram a necessidade de implementar mecanismos de *log*, pois o protótipo envolvia um processo com muitas etapas e o fato de ser *multi-thread* tornava-o muito difícil de depurar utilizando depuradores convencionais.

A implementação de um sistema de *log* permitiu detectar erros e *bugs* com mais facilidade, além de tornar possível a identificação de pontos de melhoria, como a utilização de *timeouts* ao acessar o *cache*.

Analisando-se os *logs* percebeu-se que o navegador local podia realizar requisições HTTP com maior velocidade que a velocidade de remontagem das respostas correspondentes. Para resolver o problema: elaborou-se um esquema de *timeouts* no qual, caso não fosse encontrada uma resposta no *cache*, o *proxy* aguardava um tempo determinado e tentava novamente, repetindo o processo um determinado número de vezes.

Resultado

Finalizada a implementação, foram detectadas as limitações dessa versão: algumas páginas HTML apresentaram problemas de visualização em função do *caching* de recursos pelo navegador da vítima. Além disso, páginas transmitidas com o protocolo HTTPS não foram exibidas no navegador da máquina que realizou o monitoramento.

5.3.2 Segunda versão

A motivação para a implementação da segunda versão veio das limitações encontradas na versão anterior. Dessa forma, esta versão foi elaborada desde o começo com o foco em resolver os problemas levantados, implementando assim, a segunda abordagem de filtragem de tráfego *Web*. Foram reutilizadas as ferramentas *TrotiNet* e *Selenium*, ao passo que a *PcapDotNet* foi substituída pela biblioteca *WinpkFilter* [8] que permitiu redirecionamento e manipualação de pacotes. Além disso foi adicionada a biblioteca *HtmlAgilityPack* [1] para análise e formatação de conteúdo HTML.

Desenvolvimento

O primeiro passo consistiu em trocar a abordagem de captura de pacotes pela abordagem de redirecionamento de pacotes. Inicialmente imaginou-se que uma ferramenta que fizesse NAT poderia ser configurada para realizar as manipulações propostas na seção 4.4.

Sendo assim, iniciou-se uma busca por ferramentas capazes de realizar NAT, porém foram encontrados poucas alternativas que fossem compatíveis com o *Windows*. Além disso, nenhuma das ferramentas encontradas apresentava as opções de configuração necessárias.

Chegou-se a considerar a utilização de simuladores de roteadores, porém essa hipótese foi descartada porque seria muito difícil fazer isso de maneira automatizada, além de complicar demasiadamente as configurações necessárias para instalação e uso do *Webspy*.

Em seguida, buscou-se bibliotecas que se especializassem no redirecionamento de pacotes e estivessem disponíveis para *Windows*. Foram encontradas duas: a *WinDivert* [7] e a *WinpkFilter*.

Primeiramente testou-se a biblioteca *WinDivert*. Foi construído um protótipo para verificar se a biblioteca era capaz de alterar e encaminhar pacotes como esperado. Esse protótipo foi inicialmente programado para redirecionar quaisquer pacotes recebidos em uma interface. Os testes desse protótipo foram bem sucedidos e o protótipo funcionou como esperado. Em seguida, alterou-se o protótipo para encaminhar os pacotes obtidos pelo ARP *Spoofing*.

No entanto, quando testado com o tráfego interceptado, o protótipo não foi capaz de encaminhar os pacotes alterados. O autor da biblioteca foi contatado, porém ele não conseguiu descobrir porque o protótipo não funcionava, pois, aparentemente, o código desenvolvido estava correto.

Sendo assim, decidiu-se inciar os testes com a segunda biblioteca descoberta. O *WinpkFilter*, diferentemente da ferramenta anterior, possuía bibliotecas desenvolvidas para C# o que facilitou o processo de desenvolvimento de um protótipo inicial.

Esse protótipo reaproveitou muito do código desenvolvido para a biblioteca *WinDivert*, porém, como as bibliotecas possuíam APIs muito distintas, ainda foi preciso escrever uma quantidade significativa de código adicional. O período de testes e desenvolvimento desse protótipo foi muito maior que o da *WinDivert*, pois a documentação era muito superficial.

Quando o protótipo ficou pronto, verificou-se que o ele somente funcionava na versão 8 do *Windows*. As causas ainda não foram descobertas, pois, de acordo com o fabricante da biblioteca, o *WinpkFilter* é compatível também com a versão 7 do *Windows*.

Uma vez que terminado o protótipo de redirecionamento, inciaram-se as mudanças sobre a versão anterior do módulo de visualização. A Figura 5.6 ilustra a arquitetura da nova versão proposta.

As mudanças concebidas nessa versão estão baseada na introdução do componente *Redirect* que corresponde ao protótipo implementado com a biblioteca *WinpkFilter*.

Esse componente permitiu que o tráfego interceptado pelo ARP *spoofing* pudesse ser redirecionado a um novo um servidor *proxy* (denominado *Man-in-the-Middle Proxy*) cuja função é intermediar todas as conexões do dispositivo monitorado.

Esse *proxy* também foi implementado com a biblioteca *TrotiNet* e centraliza as funções principais dessa versão:

1. Remover/alterar os cabeçalhos das mensagens HTTP recebidas e enviadas pelo *host* monitorado, de forma a tentar impedir que os recursos sejam armazenados no *cache* de seu navegador.

2. Realizar o SSL *Stripping*.

3. Armazenar transações HTTP interceptadas no *cache* e enviá-las à Pilha de Transações.

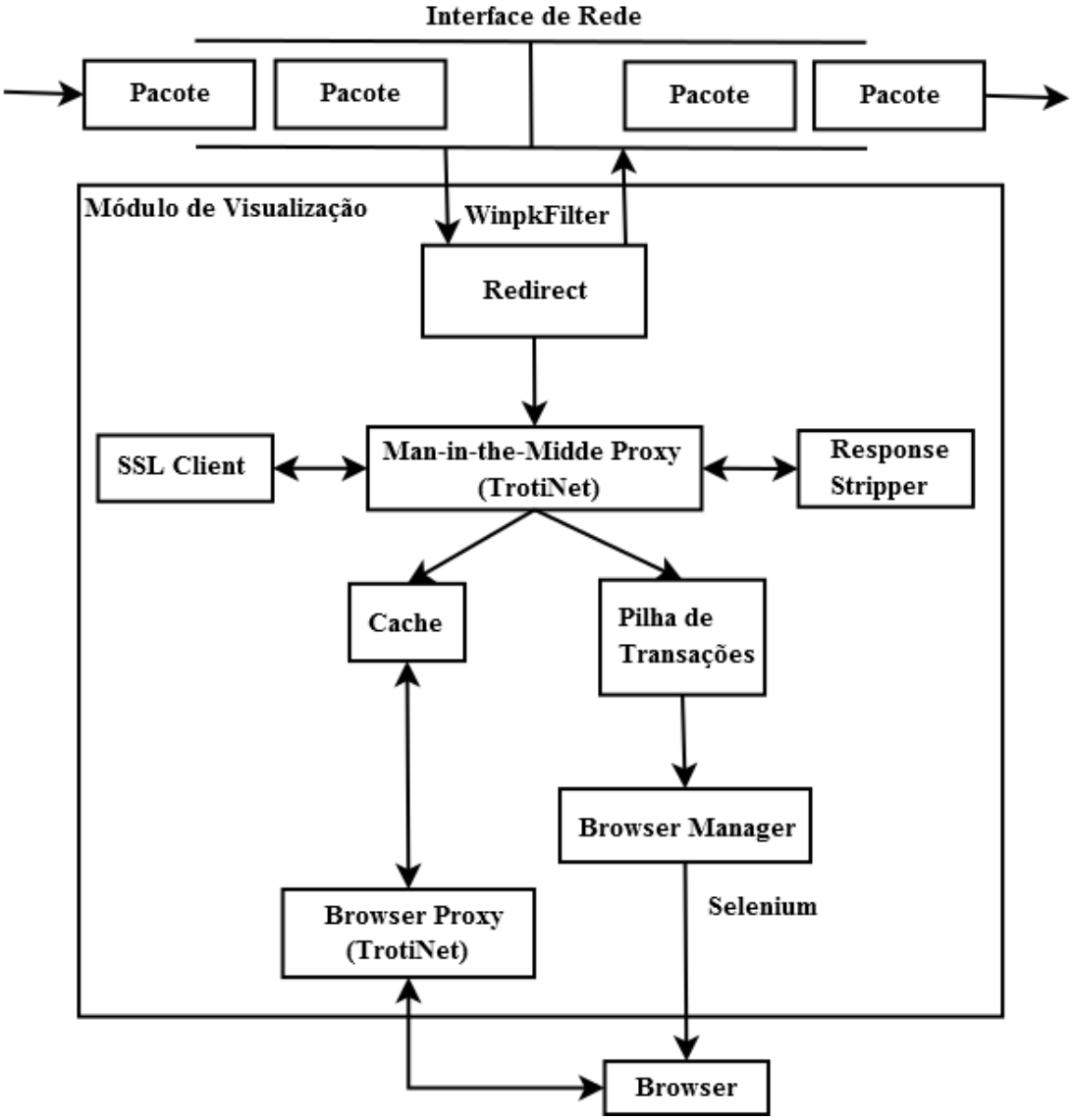


Figura 5.6: Arquitetura da implementação da segunda abordagem de filtragem

Para implementar o SSL *Stripping*, foram criados dois componentes: o SSL *Client* e o *Response Stripper*.

O SSL *Client* é responsável por estabelecer uma conexão utilizando o protocolo SSL. O processo de estabelecimento de uma conexão SSL não precisou ser implementados, pois o framework .NET já disponibilizava classes que encapsulam essa funcionalidade. No entanto, foi preciso lidar com todo o processo de envio e recepção de mensagens HTTP que, apesar de trabalhoso, não é difícil.

Durante o processo de desenvolvimento desse componente, foi preciso realizar algumas mudanças na biblioteca *TrotiNet*. Isso foi feito porque foi verificado que a biblioteca não disponibilizava formas de acessar o corpo de mensagens de requisição HTTP (o que impediu inicialmente o SSL *Client* de transmitir requisições com o método POST). Além disso, foi detectado que, por uma falha de implementação, a biblioteca era capaz de alterar apenas um pequeno conjunto cabeçalhos das mensagens HTTP. Uma vez resolvidas essas questões o desenvolvimento do componente seguiu normalmente.

O *Response Stripper*, por sua vez, é responsável por realizar modificações em mensagens HTTP, conforme descrito na seção 4.5. A abordagem elaborada inicialmente utilizava expressões regulares para identificar e realizar alterações de maneira eficiente.

No entanto, durante a realização de testes com páginas HTML variadas, verificou-se que muitas apresentam conteúdo HTML mal formatado (porém válido), o que dificultava a construção de expressões regulares que funcionassem em todos os casos possíveis.

Para resolver esse problema, foi utilizada a biblioteca *HtmlAgilityPack* capaz de realizar o *parse* de documentos HTML e formatá-los corretamente. Uma fez formatados, verificou-se que as expressões regulares eram aplicadas com sucesso.

Por fim, a nova versão adicionou também o componente denominado Pilha de Transações. Esse componente é constituido por uma pilha que recebe todas as transações cujas respostas possuem um corpo e, portanto, podem conter uma página HTML. Os componentes *Browser Manager* e *Browser Proxy* foram reutilizados sem alterações significativas.

Essa versão continuou seguindo o modelo *multi-thread* elaborado na versão anterior, porém duas *threads* foram dedicadas aos servidores *proxy* enquanto a *thread* restante ficou responsável por remover transações da Pilha de Transações e enviá-las ao *Browser Manager*.

Resultado

O problema gerado pelo *caching* de recursos realizado pelo navegador do alvo do monitoramento foi resolvido com sucesso por esta versão e não foi detectado nos testes realizados. No entanto, a visualização de páginas que utilizam o protocolo HTTPS não funcionou em todos os testes, pois não houve muito tempo para o desenvolvimento do SSL *Stripping* e, portanto, sua implementação foi bem básica. Além disso, essa técnica possui um conjunto de limitações, conforme discutido na Seção 4.5.

5.4 Módulo de *Playback*

O módulo de *Playback* tem o objetivo de reproduzir a visualização das páginas monitoradas de acordo com a ordem em que foram acessadas pelo alvo do monitoramento. Esse foi o módulo mais simples de ser implementado e também foi desenvolvido em C#.

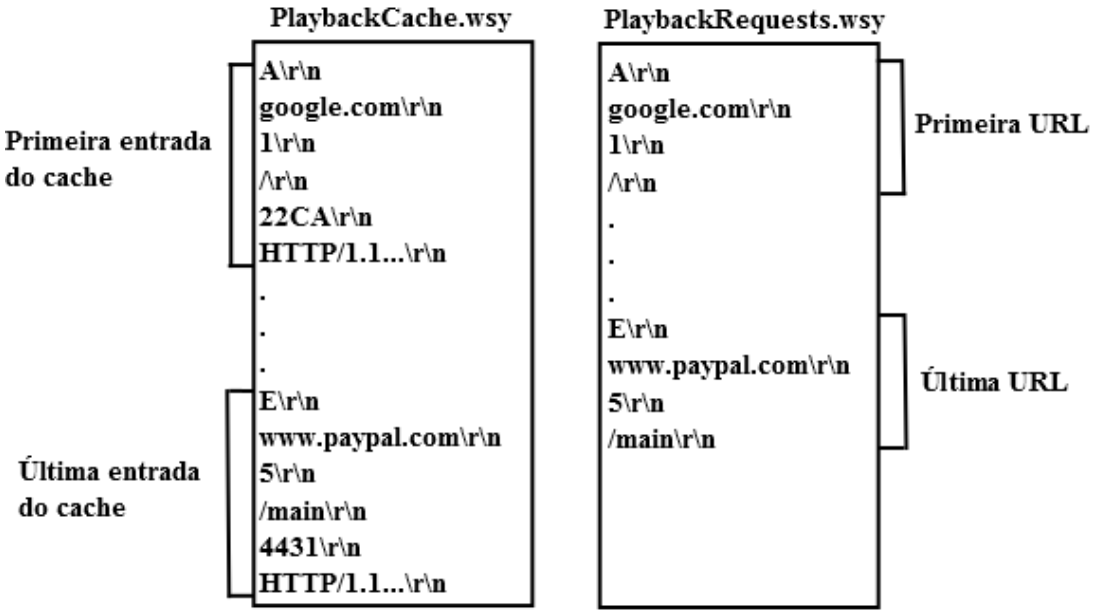


Figura 5.7: Exemplo do esquema de armazenamento dos arquivos *PlaybackCache.wsy* e *PlaybackRequests.wsy*

5.4.1 Desenvolvimento

Como o conteúdo das páginas monitoradas é proveniente, em sua maior parte, do *cache* implementado no Módulo de Visualização, foi preciso apenas implementar uma maneira de armazenar o *cache* em um arquivo e posteriormente reconstruí-lo à partir do arquivo armazenado.

Além de armazenar o *cache*, foi necessário armazenar também as URLs enviadas ao navegador *Web* local pelo componente *Browser Manager*. Os arquivos foram armazenados utilizando o padrão de codificação em *chunks* definido na versão 1.1 do protocolo HTTP, pois as rotinas de codificação e decodificação desse esquema já haviam sido implementadas.

A Figura 5.7 ilustra um exemplo que utiliza a estrutura proposta para o arquivo que armazena o *cache* (*PlaybackCache.wsy*) e o que armazena as URLs (*PlaybackRequests.wsy*).

As URLs foram armazenadas em ambos os casos divididas em *host* e URI para facilitar o processo de reconstrução.

Uma vez implementado o esquema de armazenamento/reconstrução dos arquivos, foi implementada uma rotina que recuperava o *cache* e as URLs, montava uma lista com as URLs recuperadas e por fim percorria a lista enviando cada URL ao navegador local de acordo com um intervalo informado pelo usuário.

5.4.2 Resultado

O módulo de *Playback* foi testado em conjunto com o módulo de visualização e não apresentou nenhuma limitação. Nos testes realizados, todas as páginas *Web* capturadas foram reproduzidas com sucesso.

5.5 Integração e resultados obtidos

A última etapa consistiu em integrar os módulos desenvolvidos e realizar testes de integração para que fosse alcançado o resultado final: a aplicação *Webspy*.

A integração foi iniciada pelo módulo mais complexo: o Módulo de Visualização. Em seguida, adicionou-se o módulo de ARP *Spoofing*, o o módulo de Varredura e, por fim, o de *Playback*. A integração do módulo de Varredura e de ARP *Spoofing* foi facilitada pois eles foram desenvolvidos como aplicações *standalone* e não era necessário que se comunicassem obrigatoriamente com os demais módulos.

Naturalmente surgiram diversos erros de integração, porém eles foram identificados com facilidade graças ao sistema de *log* adicionado no módulo de Visualização.

Em função da segunda versão do módulo de Visualização somente funcionar na versão 8 do *Windows*, decidiu-se separar o *Webspy* em duas versões:

0.1 Realiza captura de pacotes, armazena os dados interceptados em um arquivo de captura (.pcap) e é compatível com o *Windows* 7.

Pré-requisitos: é preciso ter instalado os componentes: NET Framework 4.0, o *driver* da biblioteca *WinPcap* e o navegador *Firefox*. Adicionalmente, a porta de número 12345 deve estar desbloqueada.

Limitações: incapaz de monitorar/exibir páginas HTTPS e a visualização de páginas interceptadas pode ser afetada pelo *caching* de recursos.

0.2 Realiza redirecionamento e manipulação de pacotes, é capaz de visualizar a maioria das páginas HTTP e implementa o SSL *stripping*.

Pré-requisitos: *Windows* 8 com os seguintes componentes instalados: NET Framework 4.0, o driver da biblioteca *WinpkFilter* e o navegador *Firefox*. Adicionalmente, as porta de número 12345 e 12346 devem estar desbloqueadas.

Limitações: a implementação do SSL *Stripping* realizada é básica e não abrangeu o tratamento de *cookies* adequado. Sendo assim, páginas que dependem de *cookies* não foram visualizadas corretamente.

Em ambas versões, os parâmetros de entrada do *Webspy* podem ser configurados por meio um arquivo de configuração que utiliza a linguagem XML como mostra a Figura 5.8. Caso o arquivo não esteja presente, o programa exibe mensagens que coletam os dados necessários.

A Figura 5.9 mostra o menu da última versão do *Webspy* na qual são exibidas 5 opções:

1. ARP *Sweep*: inicia uma varredura de rede utilizando a técnica ARP *Sweep*.
2. Ping *Sweep*: inicia uma varredura de rede utilizando a técnica Ping *Sweep*.
3. *Start Webspy*: inicia o processo de filtragem e visualização de tráfego *Web*.
4. *Replay saved session*: inicia o processo de *playback*.
5. *Exit*: fecha a aplicação.

Por fim, a Figura 5.10 apresenta o *Webspy* em funcionamento no momento que exibe uma página HTTP interceptada e a Figura 5.11 exibe a comparação entre uma página HTTPS e sua versão HTTP gerada após a realização do SSL *Stripping*.

```
<?xml version="1.0"?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
  </startup>
  <appSettings>
    <!-- Interface utilizada -->
    <add key="Interface" value=""/>
    <!-- Endereço IP da vítima-->
    <add key="VictimIP" value="192.168.1.5"/>
    <!--Habilita ou desabilita a geração dos arquivos de log-->
    <add key="Debug" value="true"/>
    <!---Configura o diretório no qual serão criados os arquivos de log-->
    <add key="LogDirectory" value=".\logs"/>
  </appSettings>
</configuration>
```

Figura 5.8: Arquivo de configuração do *Webspy*

```
*** Webspy v0.2 ***
This software is capable of redirecting intercepted pages from a device in the n
etwork to your local browser.

...


What would you like to do (1-5)?
1- Ping Sweep.
2- ARP Sweep.
3- Start Webspy.
4- Replay saved session.
5- Exit.
?
```

Figura 5.9: Opções do *menu* do *Webspy*

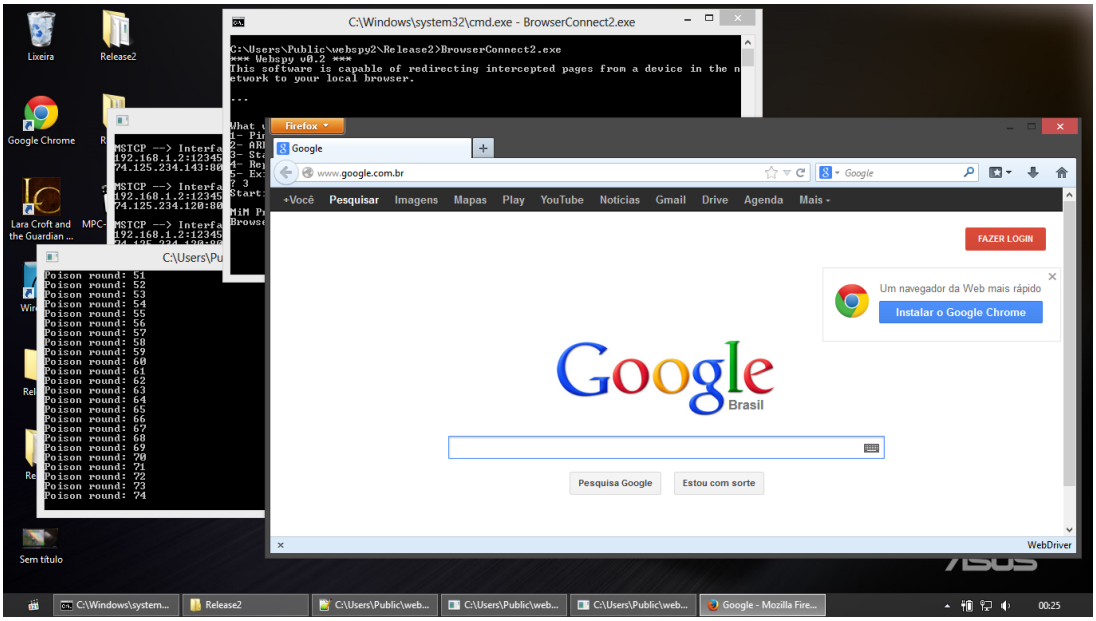
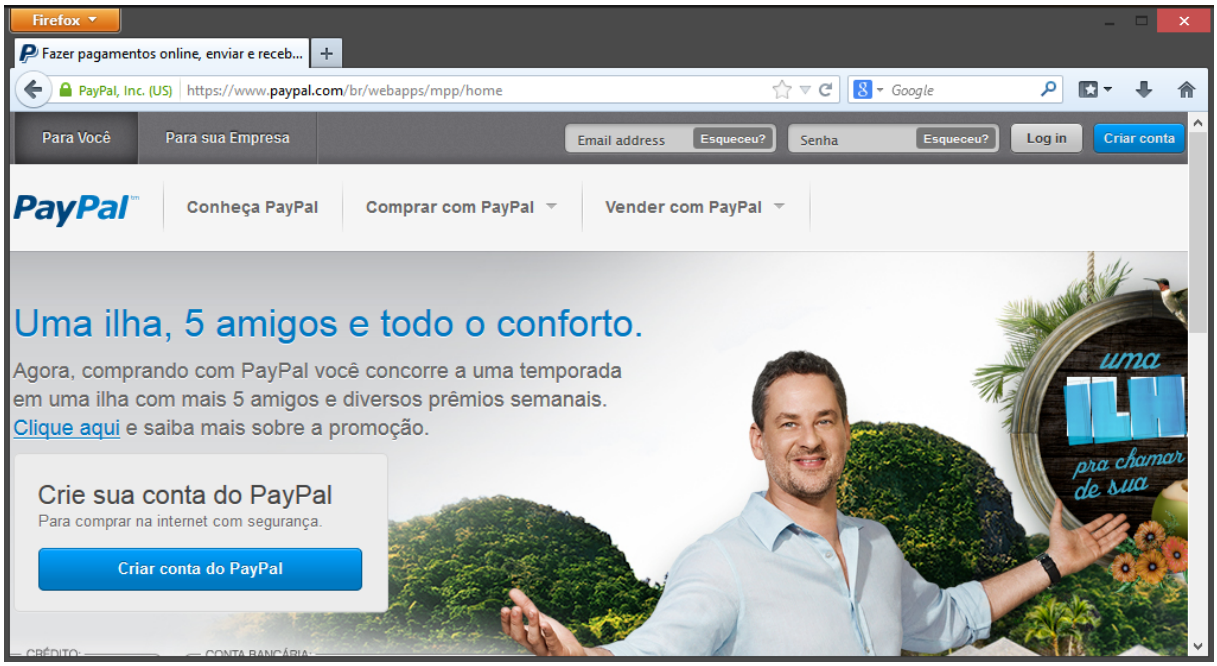
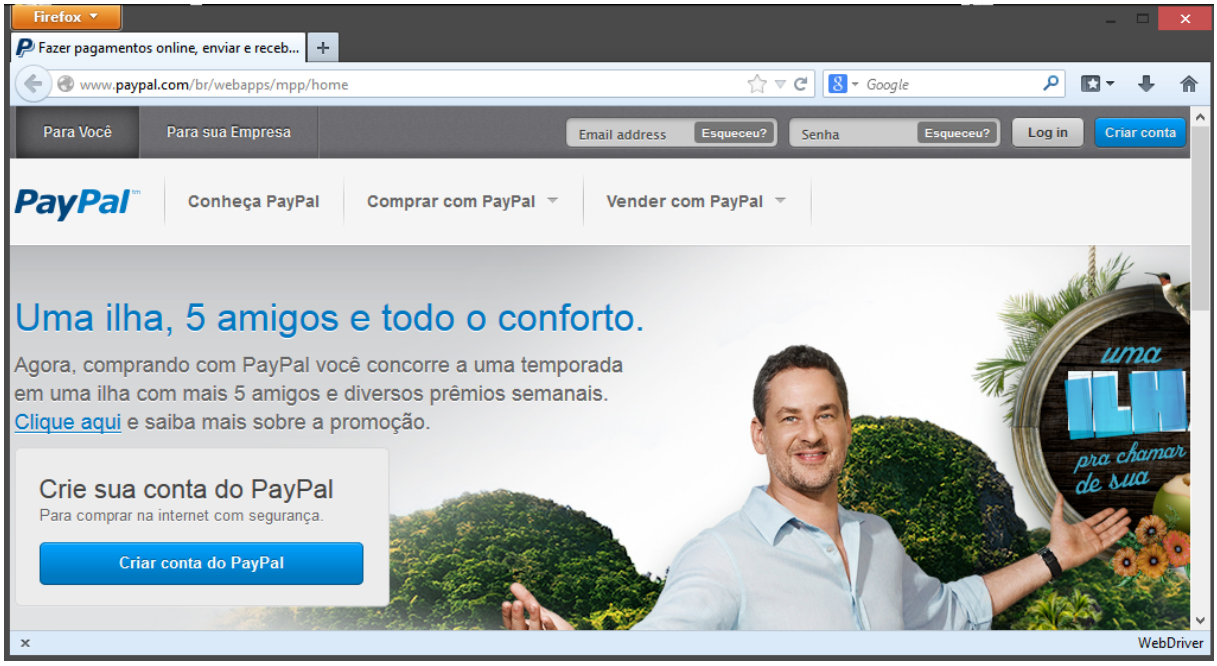


Figura 5.10: *Webspy* em funcionamento



(a) Versão legítima da página inicial do *PayPal*



(b) Versão da página inicial do *PayPal* após o SSL *Stripping*

Figura 5.11: Comparação entre uma página antes e após a realização do SSL *Stripping* pelo *Webspy*

Capítulo 6

Conclusão

O objetivo deste trabalho é discutir conceitos e teorias que permitem a construção de uma ferramenta que utiliza a técnica de ataque conhecida como ARP *Spoofing* com o objetivo de monitorar um dispositivo específico de uma rede, além de implementar e apresentar uma aplicação como prova de conceito.

Nesse aspecto, acredita-se que os objetivos foram alcançados: a ferramenta denominada *Webspy* foi implementada com sucesso e foram geradas duas versões cujas funcionalidades e limitações estão descritas na seção 5.5.

Este trabalho mostra que, apesar do ARP *Spoofing* ser uma técnica amplamente conhecida, continua sendo viável utilizá-la para realizar monitoramento (assim como ataques extremamente invasivos). Isso reflete uma despreocupação e desinformação por parte dos fabricantes de *switches* e roteadores, pois diversos métodos de contenção e detecção desse ataque já foram apresentados, como pode ser visto em [12, 29, 24].

A idéia de utilizar ferramentas de captura de pacotes para a realização de monitoramento também já foi explorada em diversas aplicações, sendo uma das mais notáveis o *Xplico* [10].

O *Xplico* é uma ferramenta *Open Source* com o foco voltado à análise forense de rede, que possui a funcionalidade de reconstrução de uma página *Web* a partir de um arquivo de captura de pacotes. A Figura 6.1 ilustra como essa ferramenta organiza os dados obtidos de uma captura.

O *Webspy* complementa as funcionalidades apresentadas por ferramentas como o *Xplico* e inova pela sua forma de visualização do tráfego monitorado em tempo real e pelo uso de uma ferramenta de automação de navegadores *Web*, que permite prover uma experiência de monitoramento que se assemelha à experiência real do usuário monitorado.

Como trabalhos futuros, pode-se citar: desenvolvimento de uma versão da aplicação voltada para sistemas *Linux*, a realização de um estudo aprofundado do SSL *Stripping* e a implementação de outras técnicas que permitem contornar o protocolo HTTPS.

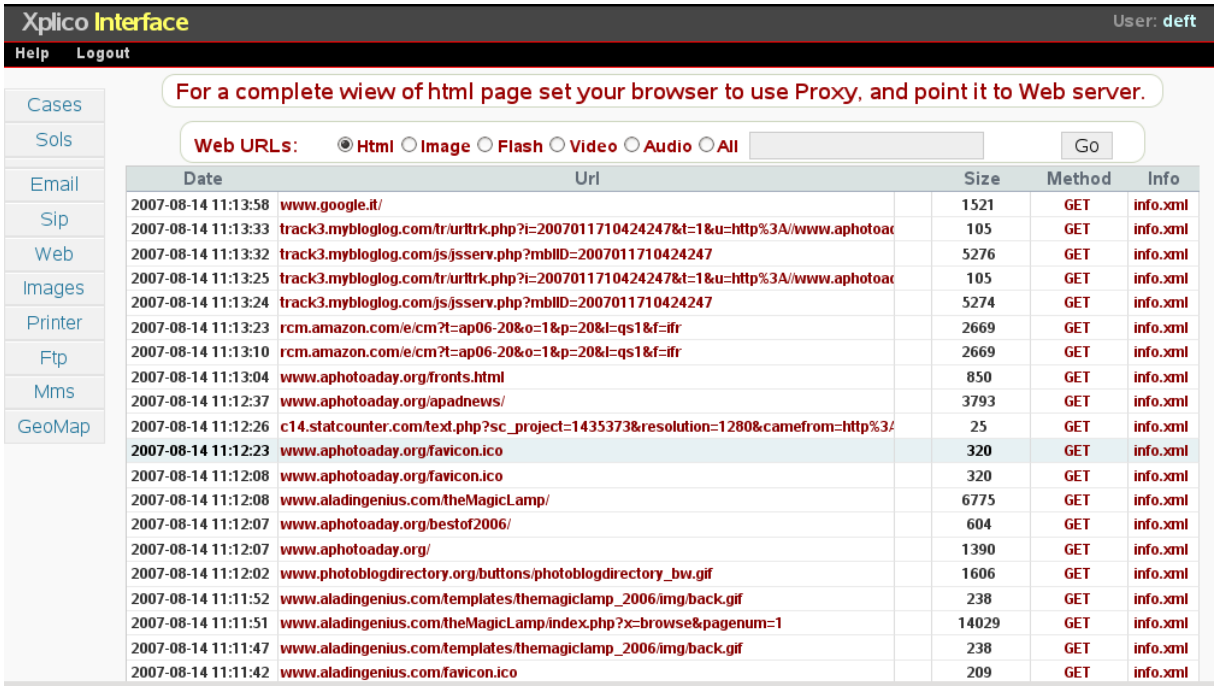


Figura 6.1: Visualização dos dados de uma captura pela ferramenta *Xplico* - retirada de [10]

Referências

[1] Html Agility Pack. Disponível em http://htmlagilitypack.codeplex.com/, Acessado em: julho de 2013. 44

[2] Pcap.Net. Disponível em http://pcapdotnet.codeplex.com/, Acessado em: julho de 2013. 41

[3] Selenium - Web Browser Automation. Disponível em http://docs.seleniumhq.org/, Acessado em: julho de 2013. 41

[4] TCPDUMP/LIBPCAP public repository. Disponível em http://www.tcpdump.org/, Acessado em: julho de 2013. 28

[5] The Libnet Packet Construction Library. Disponível em http://packetfactory.openwall.net/projects/libnet/, Acessado em: julho de 2013. 36

[6] TrotiNet: a C# HTTP proxy library. Disponível em http://trotinet.sourceforge.net/, Acessado em: julho de 2013. 41

[7] WinDivert 1.0: Windows Packet Divert. Disponível em http://reqrypt.org/windivert.html, Acessado em: julho de 2013. 45

[8] Windows Packet Filter Kit. Disponível em http://www.ntkernel.com/, Acessado em: julho de 2013. 44

[9] WinPcap: The industry-standard windows packet library. Disponível em http://www.winpcap.org/, Acessado em: julho de 2013. 36

[10] Xplico - Open Source Network Forensic Analysis Tool (NFAT). Disponível em http://www.xplico.org/, Acessado em: julho de 2013. vii, 52, 53

[11] Hayriye Altunbasak, Sven Krasser, Henry Owen, Joachim Sokol, and Jochen Grimminger. Addressing the weak link between layer 2 and layer 3 in the Internet architecture. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 417 – 418, nov. 2004. 16

[12] M. Carnut and J. Gondim. ARP spoofing detection on switched Ethernet networks: A feasibility study. In *Proceedings of the 5th Symposium on Security in Informatics*, 2003. 16, 21, 52

[13] Kefei Cheng, Meng Gao, and Ruijie Guo. Analysis and Research on HTTPS Hijacking Attacks. In *Proceedings of the 2010 Second International Conference on Networks Security, Wireless Communications and Trusted Computing - Volume 02*, NSWCTC '10, pages 223–226, Washington, DC, USA, 2010. IEEE Computer Society. 32, 34

[14] Tamara Dean. *Network+ Guide to Networks*. Course Technology Press, Boston, MA, United States, 5th edition, 2009. 8

[15] University of Delaware Dept. of Electrical Engineering. Standard for the format of Arpa Internet Text Messages, August 1982. RFC 822. 22

[16] Gary Donahue. *Network Warrior*. O'Reilly Media, Inc., 2007. vi, 8

[17] Jiang Du, Xinghui Li, and Hua Huang. A Study of Man-in-the-Middle Attack Based on SSL Certificate Interaction. In *Proceedings of the 2011 First International Conference on Instrumentation, Measurement, Computer, Communication and Control*, IMCCC '11, pages 445–448, Washington, DC, USA, 2011. IEEE Computer Society. 2

[18] Francis P. Egevang K. The IP Network Address Translator (NAT), May 1994. RFC 1631. 31

[19] Roy Fielding. Hypertext Transfer Protocol – HTTP/1.1, June 1999. RFC 2616. 28

[20] C. Horning. A Standard for the Transmission of IP Datagrams over Ethernet Networks, April 1984. RFC 894. vi, 11

[21] University of Southern California Information Sciences Institute. Internet Protocol : Darpa Internet Program Protocol Specification, September 1981. RFC 791. vi, 13

[22] Yogesh Joshi, Debabrata Das, and Subir Saha. Mitigating man in the middle attack over secure sockets layer. In *Proceedings of the 3rd IEEE international conference on Internet multimedia services architecture and applications*, IMSAA'09, pages 188–192, Piscataway, NJ, USA, 2009. IEEE Press. 2

[23] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Addison-Wesley Publishing Company, USA, 5th edition, 2009. 1

[24] Yang Liu, Kaikun Dong, Lan Dong, and Bin Li. Research of the ARP spoofing principle and a defensive algorithm. *WTOC*, 7(5):413–417, May 2008. 16, 52

[25] M. Marlingspike. New Tricks For Defeating SSL in Practice. In *BlackHat Conference*, 2009. 32

[26] Robert M. Metcalfe and David R. Boggs. Ethernet: distributed packet switching for local computer networks. *Commun. ACM*, 19(7):395–404, July 1976. 7

[27] Nick Nikiforakis, Yves Younan, and Wouter Joosen. HProxy: client-side detection of SSL stripping attacks. In *Proceedings of the 7th international conference on Detection of intrusions and malware, and vulnerability assessment*, DIMVA'10, pages 200–218, Berlin, Heidelberg, 2010. Springer-Verlag. 35

[28] Rolf Oppliger. *SSL and TLS: Theory and Practice*. Artech House, Inc., Norwood, MA, USA, 2009. viii, 62, 66

[29] S. Puangpronpitag and N. Masusai. An efficient and feasible solution to ARP Spoof problem. In *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, 2009. ECTI-CON 2009. 6th International Conference on*, volume 02, pages 910 –913, may 2009. 21, 52

[30] Somnuk Puangpronpitag and Nattavut Sriwiboon. Simple and Lightweight HTTPS Enforcement to Protect against SSL Striping Attack. In *Proceedings of the 2012 Fourth International Conference on Computational Intelligence, Communication Systems and Networks*, CICSYN '12, pages 229–234, Washington, DC, USA, 2012. IEEE Computer Society. 35

[31] Adolfo Rodriguez, John Gatrell, and Roland Peschke. *TCP/IP Tutorial and Technical Overview*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 7th edition, 2001. 5

[32] Bruce Schneier. *Schneier's Cryptography Classics Library: Applied Cryptography, Secrets and Lies, and Practical Cryptography*. Wiley Publishing, 2007. 58

[33] IEEE Computer Society. IEEE Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges, 1998. IEEE 802.1D. 8

[34] Dug Song. dsniff. Disponível em http://www.monkey.org/ dugsong/dsniff/, Acessado em: julho de 2013. 1

[35] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1993. viii, 10, 15, 25

[36] Douglas R. Stinson. *Cryptography: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1995. 58

[37] Narten T. Neighbor Discovery for IP version 6 (IPv6), September 2007. RFC 4861. 39

[38] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002. 1, 4

[39] Brian Totty, David Gourley, Marjorie Sayer, Anshu Aggarwal, and Sailu Reddy. *HTTP: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002. vi, 23, 26, 29

[40] Zouheir Trabelsi and Wassim El-Hajj. ARP spoofing: a comparative study for education purposes. In *2009 Information Security Curriculum Development Conference*, InfoSecCD '09, pages 60–66, New York, NY, USA, 2009. ACM. 16

[41] Zouheir Trabelsi and Wassim El-Hajj. On investigating ARP spoofing security solutions. *Int. J. Internet Protoc. Technol.*, 5(1/2):92–100, April 2010. 21

[42] Ruben Unteregger. APE – The ARP Poisoning Engine. Disponível em http://www.megapanzer.com/2012/04/11/ape-the-arp-poisoning-engine/, Acessado em: julho de 2013. 39

[43] Geon Yoon, Dae Hyun Kwon, Soon Chang Kwon, Yong Oon Park, and Young Joon Lee. Ring Topology-based Redundancy Ethernet for Industrial Network. In *SICE-ICASE, 2006. International Joint Conference*, pages 1404 –1407, oct. 2006. 7

[44] Sendong Zhao, Ding Wang, Sicheng Zhao, Wu Yang, and Chunguang Ma. Cookie-proxy: a scheme to prevent SSLStrip attack. In *Proceedings of the 14th international conference on Information and Communications Security*, ICICS'12, pages 365–372, Berlin, Heidelberg, 2012. Springer-Verlag. 35

[45] H. Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *Communications, IEEE Transactions on*, 28(4):425 – 432, apr 1980. 5

Apêndice A

Criptografia e protocolo SSL

As técnicas descritas no capítulo 3.4 permitem interceptar e visualizar de maneira clara o tráfego de rede de um determinado *host* de uma rede. Porém, existe uma premissa básica que torna isso possível: as informações são transmitidas e recebidas em claro, ou seja, sem a utilização de nenhuma técnica de criptografia. Este capítulo apresenta os conceitos básicos de criptografia e do protocolo SSL.

A.1 Conceitos de Criptografia

O objetivo principal da criptografia, de acordo com [36, 32], é permitir que duas pessoas ou entidades possam comunicar-se por meio de um canal inseguro de tal forma que uma terceira pessoa considerada um oponente não consiga compreender as informações transmitidas. Uma informação que pode ser compreendida por qualquer pessoa é denomidada texto em claro.

A idéia da criptografia é transformar uma mensagem representada como texto em claro em um criptograma, isto é, um conjunto de caracteres cujo significado não pode ser extraído sem alguma informação adicional (tipicamente uma chave). Esse processo é chamado de cifragem e, para que a criptografia seja de fato útil, é necessário possibilitar também a realização do processo contrário: a decifragem.

De maneira mais formal, considerando M como o espaço de possíveis mensagens, C o espaço de criptogramas e K o espaço de chaves, podemos definir o conceito de cifra:

- Uma cifra é um conjunto K de funções simbólicas inversíveis $e : M \rightarrow C$ e $e^{-1} : C \rightarrow M$ tal que:
  - $m \in M$ codifica um texto em uma linguagem L.
  - $\forall m \in M, \forall e, e^{-1} \in K$ tem-se que $e(m) = c$ e $e^{-1}(c) = m$ com $c \in C$.
  - e e e^{-1} podem ser parametrizadas por uma chave compartilhada k ou por um par k_{PUB}, k_{PRIV}.

Uma vez definida a cifra, pode-se definir um algoritmo de cifra: uma implementação f de uma cifra K tal que $f : K \times M \leftrightarrow K \times C$, onde:

- f cifra uma mensagem m, isto é, calcula $e(m) = c$.

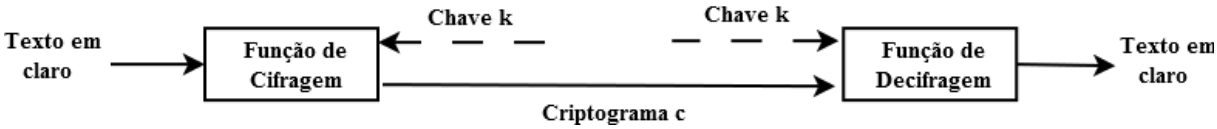


Figura A.1: Esquema simétrico aplicado à cifragem de mensagens

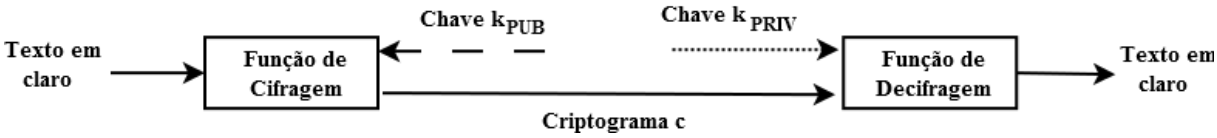


Figura A.2: Esquema assimétrico aplicado à cifragem de mensagens

- f decifra um criptograma c, isto é, calcula $e^{-1}(m) = m$.

Uma cifra robusta é aquela cujo custo para se obter a mensagem m a partir de c sem o conhecimento da chave k é inviável. Os algoritmos de cifra, por sua vez, são divididos em três: restritos, simétricos e assimétricos. O restrito é aquele cuja robustez da cifra depende do sigilo do projeto da função f e das chaves caso existam. Por sua vez, a robustez da cifra do algoritmo simétrico não depende do sigilo do projeto de f, mas sim do sigilo da chave k utilizada para cifrar e decifrar, da aleatoriedade da escolha da chave e do espaço de chaves K ser grande.

O algoritmo assimétrico, por fim, possui duas chaves: k_{PUB}, a chave pública e k_{PRIV}, a chave privada. Nesse esquema de cifra, a robustez também não depende do sigilo do projeto de f nem de k_{PUB} e ele é construído de forma a tornar inviável a dedução de k_{PRIV} a partir de k_{PUB}. A robustez, depende então do sigilo de k_{PRIV}, da equiprobabilidade da escolha do par (k_{PUB},k_{PRIV}) no espaço de chaves K e desse espaço ser suficientemente grande.

As Figuras A.1 e A.2 demonstram como podem ser utilizados os esquemas simétrico e assimétrico, respectivamente, no processo de cifragem de mensagens. O esquema simétrico exige o sigilo da chave k entre duas pessoas apenas e a sua integridade durante seu uso, pois o remetente irá utilizá-la para cifrar uma mensagem e o destinatário precisará da mesma chave para decifrar a mensagem.

Por outro lado, o assimétrico exige o sigilo apenas da chave k_{PRIV} e a integridade das duas chaves em sua utilização. A chave pública k_{PUB} será utilizada por qualquer remetente que necessitar transmitir mensagens para o destinatário, que, por sua vez, irá utilizar a chave privada k_{PRIV} para decifrar a mensagem. O fato da chave privada ser de posse exclusiva do destinatário e ela não poder ser derivada a partir da chave pública em tempo viável garante que uma terceira pessoa que tenha acesso ao criptograma não terá acesso ao conteúdo da mensagem que foi transmitida.

A.2 Certificados e assinatura digital

Além do problema de transmissão sigilosa, o esquema assimétrico é utilizado para implementar um esquema de certificação e assinatura digital. O objetivo principal do

certificado é atestar um conjunto de informações sobre uma pessoa. Dessa forma, um certificado assemelha-se muito com um documento de identidade. Um documento desse tipo possui informações como nome, número de registro, endereço etc. O ponto chave é como determinar que as informações contidas nesse documento são de fato verdadeiras.

Esses documentos são emitidos por um órgão ou entidade que assume-se confiável (porém cada pessoa tem a liberdade de aceitar ou não esta premissa) e possuem mecanismos que validam sua integridade como, por exemplo, uma marca d'água. O processo de certificação digital é semelhante. As entidades capazes de emitir certificados são chamadas de Autoridades Certificadoras (ACs).

Uma pessoa que deseja possuir um certificado, aqui denominado como requerente, deve contatar uma AC, fornecer seus dados e seguir as regras definidas na Política de Certificação da AC. Essas regras definem os procedimentos (ou a ausência deles) que a AC utiliza para verificar/validar os dados apresentados por um usuário durante a requisição de seu certificado.

O próximo passo, de acordo com o esquema de certificação, consiste na geração de um par de chaves k_{PUB} e k_{PRIV} que serão colocados no certificado juntamente com as informações da AC e uma assinatura digital sobre todo o conteúdo reunido no certificado. Como a maioria dos requerentes não possuem o conhecimento técnico para gerar seu próprio par de chaves nem entendem as possíveis consequências disto, esta tarefa normalmente é delegada à própria AC.

Este detalhe constitui uma falha grave das premissas do esquema assimétrico, pois a AC terá acesso à chave privada do requerente (podendo assim armazená-la). Dessa forma, o sigilo da chave privada não pode ser garantido, o que destrói toda a segurança e confiabilidade que o esquema deveria prover. No entanto, supõe-se, neste trabalho, que ocorre o caso ideal no qual o usuário gera seu próprio par de chaves.

O esquema de assinatura digital, por sua vez, busca estabelecer em documentos digitais as mesmas funções semiológicas que a assinatura de punho realiza em documentos físicos.

1. Inforjabilidade: uma entidade verificadora consegue determinar a autoria de uma assinatura em um documento.
2. Inviolabilidade: confiança da entidade verificadora quanto à integridade de um documento vinculado a uma assinatura.
3. Irrecuperabilidade: confiança na inviabilidade do reuso de uma assinatura de um documento em outro.
4. Irrefutabilidade: confiança na inviabilidade da negação de autoria da assinatura pelo autor.

Para assinar digitalmente um documento, é necessário que o signatário possua um certificado digital. Com ele, o titular é capaz de assinar o documento utilizando sua chave privada, cuja posse é exclusivamente sua e qualquer pessoa que tenha acesso à sua chave pública pode verificar se a assinatura é válida ou não. É importante lembrar, que o que é de fato assinado é o *hash* do documento e não o documento em si. Isso é feito para garantir que não seja possível alterar o documento após ele ter sido assinado.

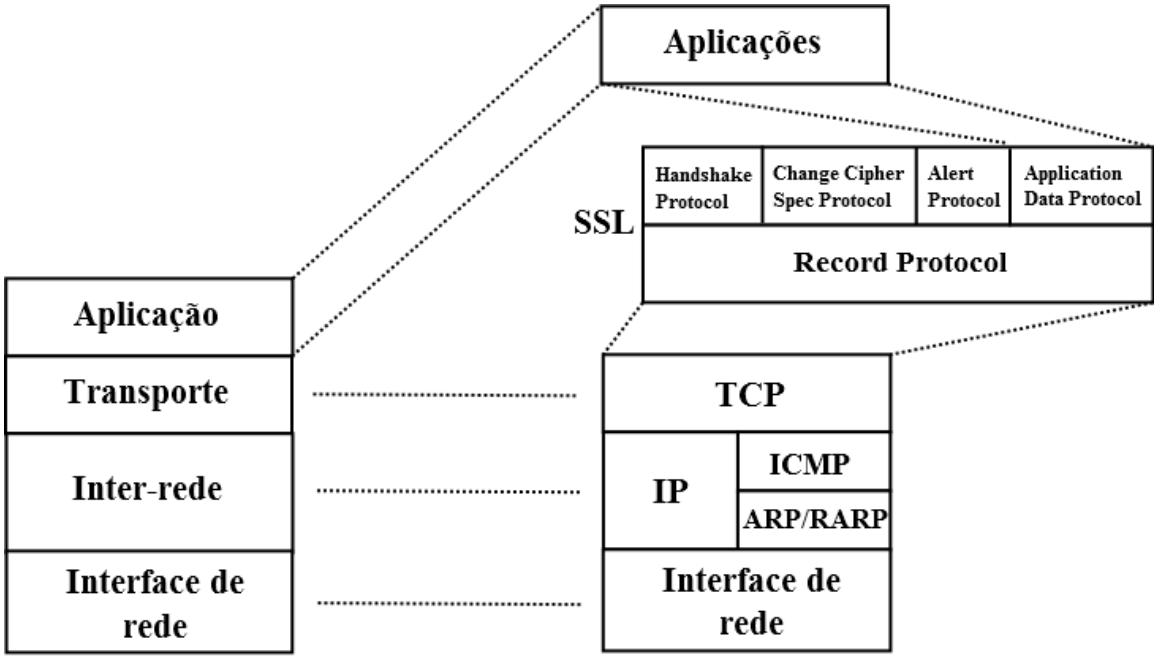


Figura A.3: Visão geral sobre a estrutura do protocolo SSL

A.3 Protocolo SSL

O protocolo SSL (*Secure Sockets Layer*) tem o objetivo de prover privacidade, autenticidade e integridade de origem entre duas aplicações que desejam estabelecer uma conexão. Ele funciona como uma camada intermediária entre a camada de transporte e a de aplicação como mostra a Figura A.3.

O SSL é na verdade um conjunto de 5 protocolos distribuídos em 2 níveis: *SSL Record Protocol* no nível 1 e *SSL Handshake Protocol*, *SSL Change Cipher Spec Protocol*, *SSL Alert Protocol* e *Application Data Protocol* no nível 2.

A.3.1 *SSL Record Protocol*

O *SSL Record Protocol* é utilizado para encapsulamento dos dados dos protocolos de nível 2 do SSL. O primeiro passo realizado consiste na divisão dos dados em blocos de 2^{14} *bytes* (ou menos) denominadas fragmentos. Cada bloco é então empacotado em uma estrutura denominada `SSLPlainText`.

Em seguida, a estrutura `SSLPlainText` é comprimida de acordo com o método de compressão especificado no estado da sessão SSL. Na verdade, somente será realizada a compressão caso um método tenha sido de fato selecionado, o que geralmente não acontece. Após a compressão, a estrutura `SSLPlainText` é convertida na estrutura `SSLCompressed` que será encriptada de acordo com uma configuração da sessão SSL denominada *cipher spec*.

Essa configuração define um par de algoritmos que serão utilizados para realizar autenticação (cálculo de *hash*) e cifragem dos dados. Juntamente com o algoritmo de troca

Tabela A.1: *Cipher suites* do protocolo SSL e os algoritmos correspondentes - adaptada de [28]

Nome	Troca de Chaves	Cifragem	Autenticação
SSL_NULL_WITH_NULL_NULL	NULL	NULL	NULL
SSL_RSA_WITH_NULL_MD5	RSA	NULL	MD5
SSL_RSA_WITH_NULL_SHA	RSA	NULL	SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5	RSA_EXPORT	RC4_40	MD5
SSL_RSA_WITH_RC4_128_MD5	RSA	RC4_128	MD5
SSL_RSA_WITH_RC4_128_SHA	RSA	RC4_128	SHA
SSL_RSA_EXPORT_WITH_RC2_CBC_40 MD5	RSA_EXPORT	RC2_CBC_40	MD5
SSL_RSA_WITH_IDEA_CBC_SHA	RSA	IDEA_CBC	SHA
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA	EXPORT_DES40_CBC	SHA
SSL_RSA_WITH_DES_CBC_SHA	RSA	DES_CBC	SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA	RSA	3DES_EDE_CBC	SHA
SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	DH_DSS_EXPORT	DES40_CBC	SHA
SSL_DH_DSS_WITH_DES_CBC_SHA	DH_DSS	DES_CBC	SHA
SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA	DH_DSS	3DES_EDE_CBC	SHA
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	DH_RSA_EXPORT	DES40_CBC	SHA
SSL_DH_RSA_WITH_DES_CBC_SHA	DH_RSA	DES_CBC	SHA
SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA	DH_RSA	3DES_EDE_CBC	SHA
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	DHE_DSS_EXPORT	DES40_CBC	SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA	DHE_DSS	DES_CBC	SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DHE_DSS	3DES_EDE_CBC	SHA
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	DHE_RSA_EXPORT	DES40_CBC	SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA	DHE_RSA	DES_CBC	SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA	DHE_RSA	3DES_EDE_CBC	SHA
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5	DH_anon_EXPORT	RC4_40	MD5
SSL_DH_anon_WITH_RC4_128_MD5	DH_anon	RC4_128	MD5
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA	DH_anon	DES40_CBC	SHA
SSL_DH_anon_WITH_DES_CBC_SHA	DH_anon	DES_CBC	SHA
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA	DH_anon	3DES_EDE_CBC	SHA
SSL_FORTEZZA_KEA_WITH_NULL_SHA	FORTEZZA_KEA	NULL	SHA
SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA	FORTEZZA_KEA	FORTEZZA_CBC	SHA
SSL_FORTEZZA_KEA_WITH_RC4_128_SHA	FORTEZZA_KEA	RC4_128	SHA

de chaves, essas configurações definem uma *cipher suite*. São definidas 31 combinações de *cipher suite* como mostra a tabela A.1.

Uma vez obtida a estrutura comprimida `SSLCompressed`, é calculado o autenticador MAC da mensagem utilizando a função *hash* definida na *cipher suite*. O MAC e a estrutura comprimida são então cifrados gerando a estrutura `SSLCiphertext`. A última etapa consiste na adição de um cabeçalho à estrutura cifrada que contém 3 campos como mostra a Figura A.5:

1. Tipo: indica a qual protocolo de nível 2 do SSL se referem os dados da estrutura.
2. Versão: indica qual versão do protocolo SSL está sendo usada.
3. Tamanho: indica o comprimento em *bytes* da estrutura `SSLCiphertext`.

O processo completo realizado pelo *SSL Record Protocol* é ilustrado na Figura A.4.

A.3.2 *SSL Handshake Protocol*

O *SSL Handshake Protocol* permite que cliente e servidor possam se autenticar e negociar configurações como métodos de compressão e *cipher suites*. Uma nova conexão se dá da seguinte maneira:

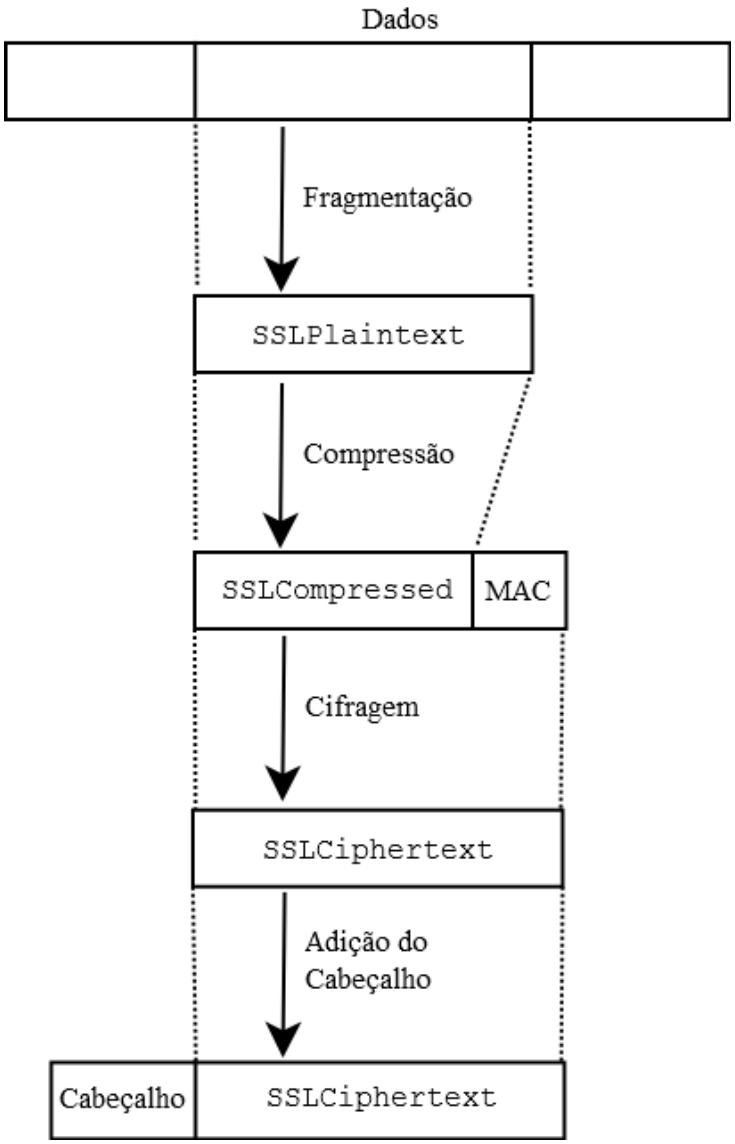


Figura A.4: Etapas do processo realizado pelo *SSL Record Protocol*

Figura A.5: Campos do cabeçalho que encapsula a estrutura `SSLCiphertext`

1. O cliente gera uma cadeia de *bits* aleatória de 28 *bytes* e concatena-a com uma *string* que define a data atual em segundos, formando assim o campo `CLIENTHELLO.random`. Em seguida, adiciona a cadeia gerada e uma lista de opções de *cipher suite* suportadas pelo cliente a uma mensagem `CLIENTHELLO` e a envia para o servidor.

2. Após receber uma mensagem `CLIENTHELLO`, o servidor envia para o cliente:

   (a) Uma mensagem `SERVERHELLO` que contém a *string* aleatória do servidor `SERVERHELLO.random` (gerada da mesma forma que a do cliente), a opção de *cipher suite* escolhida pelo servidor e um número de ID da sessão.

   (b) Caso o servidor deseje autenticar-se (o que geralmente acontece), uma mensagem `CERTIFICATE` que contém o certificado de chave pública do servidor e sua cadeia de certificados correspondente.

   (c) Caso o servidor exija que o cliente se autentique com um certificado, uma mensagem `CERTIFICATEREQUEST` que contém os tipos de certificados que são aceitos e a lista de ACs consideradas válidas pelo servidor.

   (d) Uma mensagem `SERVERHELLODONE` indicando que o servidor já enviou todas mensagens que sucedem o `SERVERHELLO`.

3. O cliente envia para o servidor:

   (a) Caso tenha recebido uma mensagem `CERTIFICATEREQUEST`, envia uma mensagem `CERTIFICATE` semelhante à que recebeu do servidor, porém com o seu certificado de chave pública e sua cadeia.

   (b) Uma mensagem `CLIENTKEYEXCHANGE` que contém o *pre-master secret*, um número aleatório de 46 *bytes* cifrado utilizando a chave pública do servidor. Esse número será decifrado pelo servidor para gerar o *master secret* que, por sua vez, será utilizado para gerar as chaves de sessão.

   (c) Caso o cliente tenha enviado uma mensagem `CERTIFICATE`, uma mensagem `CERTIFICATEVERIFY` assinada digitalmente com a chave privada correspondente à chave pública presente no certificado enviado. Essa mensagem serve como prova de que o cliente de fato tem posse da chave privada.

   (d) Uma mensagem `CHANGECIPHERSPEC` como especificado pelo *SSL Change Cipher Spec Protocol*.

   (e) Uma mensagem `FINISHED` para verificar que a troca de chaves e os processos de autenticação foram bem sucedidos. Esta mensagem é a primeira (do lado do cliente) a ser enviada cifrada de acordo com a *cipher suite* escolhida.

4. O servidor envia o último conjunto de mensagens:

   (a) Uma mensagem `CHANGECIPHERSPEC` como especificado pelo *SSL Change Cipher Spec Protocol*.

   (b) Uma mensagem `FINISHED`. Esta mensagem é a primeira (do lado do servidor) a ser enviada cifrada de acordo com a *cipher suite* escolhida.

Uma vez finalizado o *handshake*, cliente e servidor podem enviar dados de aplicação. O processo para retomar uma conexão já existente é uma simplificação desse processo:

1. O cliente envia para o servidor uma mensagem `CLIENTHELLO`, porém adiciona a ela o ID de sessão da conexão.
2. O servidor recebe a mensagem e verifica se existe uma sessão com o ID recebido em seu *cache* de sessão. Se houver e o servidor estiver disposto a reestabelecer a conexão retomando seu estado, ele envia:

   (a) Uma mensagem `SERVERHELLO` com o ID da sessão.

   (b) Uma mensagem `CHANGECIPHERSPEC`.

   (c) Uma mensagem `FINISHED`.

   Caso contrário, o servidor gera um novo ID de sessão e o processo de *handshake* completo deve ser realizado novamente.

A.3.3 *SSL Change Cipher Spec Protocol*

O objetivo desse protocolo é sinalizar transições de estratégias de cifragem entre duas entidades que estão se comunicando. Para isso, o protocolo utiliza a mensagem `CHANGECIPHERSPEC` que indica a *cipher spec* que será utilizada nas mensagens subsequentes. Esse protocolo permite que o estado da conexão SSL seja alterado sem que seja necessário realizar uma renegociação da conexão.

A.3.4 *SSL Alert Protocol*

O *SSL Alert Protocol* permite que as entidades comunicantes possam trocar mensagens de alerta. Cada mensagem possui uma descrição e um nível de alerta como mostra a tabela A.2:

Essas mensagens são utilizadas para detecção e tratamento de erros durante a conexão SSL, sendo que alguns deles (os erros fatais) ocasionam o término da sessão.

A.3.5 *SSL Application Data Protocol*

Esse protocolo é responsável pela transmissão de dados de aplicação entre entidades comunicantes. O protocolo lida com a coleta e envio dos dados de aplicação para o *SSL Record Protocol* e vice-versa.

Tabela A.2: Mensagens de alerta do *SSL Alert Protocol* - adaptada de [28]

Alerta	Código	Nível	Descrição
close_notify	0	Aviso	Remente notifica o destinatário que não enviará mais mensagens durante a conexão.
unexpected_message	10	Fatal	Remente notifica o destinatário que uma mensagem inesperada foi recebida.
bad_record_mac	20	Fatal	Remente notifica o destinatário que uma mensagem com MAC incorreto foi recebida.
decompression_failure	30	Fatal	Remente notifica o destinatário que a função de descompressão recebeu dados de entrada impróprios.
handshake_failure	40	Fatal	Remente notifica o destinatário que não foi possível negociar um conjunto parâmetros de segurança viáveis dadas as opções apresentadas.
no_certificate	41	Aviso	Cliente notifica o servidor que não possui um certificado que possa satisfazer a requisição do servidor.
bad_certificate	42	Aviso	Remente notifica o destinatário que o certificado recebido está corrompido.
unsupported_certificate	43	Aviso	Remente notifica o destinatário que o certificado recebido não é suportado.
certificate_revoked	44	Aviso	Remente notifica o destinatário que o certificado recebido foi revogado pela AC que o emitiu.
certificate_expired	45	Aviso	Remente notifica o destinatário que o certificado recebido expirou.
certificate_unknown	46	Aviso	Remente notifica o destinatário que ocorreu algum problema durante o processamento do certificado recebido.
illegal_parameter	47	Fatal	Remente notifica o destinatário que algum campo da mensagem de *handshake* se encontra inconsistente.