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

Monitoramento e Visualização de Navegação Web em Tempo Real

Felipe Rodopoulos de Oliveira

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

Orientador
Prof. M.Sc. João José Costa Gondim

Brasília
2015

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

Coordenador: Prof. Dr. Homero Luiz Piccolo

Banca examinadora composta por:

Prof. M.Sc. João José Costa Gondim (Orientador) — CIC/UnB
Prof. Dr. André Costa Drummond — CIC/UnB
Prof. Dr. Robson de Oliveira Albuquerque — ENE/UnB

CIP — Catalogação Internacional na Publicação

de Oliveira, Felipe Rodopoulos.

Monitoramento e Visualização de Navegação Web em Tempo Real / Felipe Rodopoulos de Oliveira. Brasília : UnB, 2015.
195 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2015.

1. *man-in-the-middle*, 2. *arp spoofing*, 3. *ssl stripping*, 4. *webspy*

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

Monitoramento e Visualização de Navegação Web em Tempo Real

Felipe Rodopoulos de Oliveira

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

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

Prof. Dr. André Costa Drummond
CIC/UnB

Prof. Dr. Robson de Oliveira Albuquerque
ENE/UnB

Prof. Dr. Homero Luiz Piccolo
Coordenador do Bacharelado em Ciência da Computação

Brasília, 12 de Agosto de 2015

Dedicatória

Dedico este trabalho a minha mãe, a quem sempre se mostrou preocupada com o andamento do mesmo.

Também dedico especialmente a Alexandre Dantas, o Kure, que se mostrou um dos cientístas da computação mais entusiasmados que já conheci e infelizmente nos deixou de forma repentina. Que todo seu entusiasmo possa inspirar as pessoas, assim como me inspirou.

Finalmente, também dedico este a todos os cientístas da computação sérios, principalmente meus colégas de curso. São estes que sabem que este curso não é trivial e completar o mesmo, com dedicação, é uma vitória com honra.

Agradecimentos

Inicialmente agradeço imensamente a minha mãe, a quem eu devo toda a minha base emocional e educacional. Não há ninguém que possa te substituir. Também, agradeço a minha família: meu pai, João, Sara, Osias, meus avós Carlos e Sônia, e todos os outros por estar sempre preocupados com comigo e se mostrar um refúgio de confiança e serinidade.

Agradeço o meu orientador, João Gondim, por ter me apresentado o quão fantástica a área de redes é, mostrando que qualquer conteúdo pode nos trazer um pouco de diversão.

Um grande obrigado a Annelise, por ter sido meu pilar nos últimos anos e me dar todo o apoio que eu precisei, da forma mais altruísta que já vi. Agradeço também ao Matheus Pimenta, pois sem ele, este trabalho não seria feito com a excelência desejada e, ao mesmo tempo, se mostrou um amigo fiel e verdadeiro.

Agradeço aos meus colegas de curso, pois estes me mostraram que numa universidade, não existe uma formação solitária e sim repleta de bons momentos a serem compartilhados com amigos. Portanto, Heitor, Filipe Caldas, Wallace, Paulo Alfonso, Ítalo, André, Jaqueline e todos os demais: obrigado, de verdade.

Finalmente, mas não menos importante, um obrigado fundamental a todos os meus amigos, desde épocas anteriores, até os que fiz na UnB. Graças a vocês, eu pude me manter são e feliz, além de terem me proporcionado a melhor juventude que poderia imaginar.

Resumo

Um ataque de monitoramento e visualização de navegação *web* de um *host* em tempo real consiste em interceptar o tráfego entre um alvo e seu respectivo *gateway*, utilizando os dados para renderizar as páginas da *web* correspondentes em um navegador local. Este procedimento já foi abordado em trabalhos anteriores [36] [18], recebendo o nome de Webspy, tendo como base o ARP *Spoofing* para o desvio de tráfego. Entretanto, estes trabalhos já não dão resultados efetivos, uma vez que foram ambientados de redes cabeadas, com transporte de dados em canais inseguros e páginas da *web* estáticas, com poucos arquivos. Atualmente, para um ataque decente, é necessário também levar em conta as redes sem fio, uso dos protocolos SSL/TLS, para cifragem da comunicação entre cliente e servidor, e o dinamismo presente na *web*, produzido pelo JavaScript e utilização de *cookies* do HTTP. Este trabalho tem como objetivo o tratamento destes pontos, explorando a técnica do SSL *Stripping* e elaborando esquemas para solucionar o tratamento de *cookies* e sessões, assim como as constantes requisições assíncronas, viáveis pelo JavaScript.

Palavras-chave: *man-in-the-middle*, *arp spoofing*, *ssl stripping*, *webspy*

Abstract

A host's web traffic monitoring and real-time visualization attack consists in intercept the traffic between a victim and its gateway, using its data to render the corresponding web pages in a local web browser. This procedure has already bem addressed in previous works [36] [18], being called Webspy, based on the ARP Spoofing technique. However, these works no longer provide effective results, once they were implemented in wired networks, with data transport in insecure links and static web pages, with few files. Currently, for a decent attack, it's a must to deal with wireless networks, use of SSL/TLS protocols, for the communication encryption between client and server, and the current dynamism found in the web, produced by Javascript and use of HTTP cookies. This project aims to deal with these matters, exploring the SSL Stripping technique and developing solutions to treat cookies, sessions, and asynchronous requests caused by JavaScript as well.

Keywords: *man-in-the-middle*, *arp spoofing*, *ssl stripping*, *webspy*

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O surgimento das redes de computadores foi um grande marco para a computação, transformando-se numa extensa área de pesquisa, com vasta literatura a ela dedicada[39] [16] [11]. Como qualquer outra área da Ciência da Computação, esta possui aspectos voltados à segurança. Uma vez que arquiteturas, protocolos e padrões foram projetados para uma dada situação espacial e temporal, acabam por não prever certas conjunturas, oferecendo algumas brechas que possibilitam violações de segurança e quebras de regras. Isto envolve tanto os dados trafegados pela rede como a integridade física e operacional dos dispositivos.

Essas características não são antigas. Ataques em redes existem há muito tempo e exploram diversos níveis operacionais da mesma, ou seja, das camadas de protocolos e suas integrações. Cada ataque visa um objetivo, podendo explorar uma ou mais camadas de abstração da rede.

Obviamente, cientistas e peritos da área criaram técnicas, aplicações e estratégias para prevenirem esses ataques. Várias destas são aplicadas hoje em dia e, consequentemente, vários ataques se deram por antiquados e inviáveis. Entretanto, para os mesmos se tornarem novamente funcionais, basta uma nova abordagem ou até algum pequeno retoque e novamente temos uma ameaça funcional.

Em trabalhos antigos [18] [36] foi apresentada uma ferramenta de cunho interessante: um interceptador de tráfego HTTP, que permitia um atacante visualizar uma sessão *web* de uma dada vítima, sendo apelidado de *Webspy*. Entretanto, a ferramenta se limitava a certos contextos: redes cabeada, com um suporte ineficiente a tráfego cifrado por SSL/TLS, renderização das telas ignorando requisições assíncronas, e assim por diante. Analisando, conclui-se que este não é mais o único ambiente notável a ser considerado para o contexo de troca de informações pela *web* pois, atualmente, há um grande uso da tecnologia *Wi-Fi*, da proteção de tráfego cifrado por protocolos competentes, como o TLS e o uso constante de requisições assíncronas ao servidor, através do *JavaScript*.

Logo, é necessária uma adaptação da ferramenta inicial aos moldes de acesso da *web* atual, incluindo modificações em uma ferramenta antiga, voltada à redes antigas, para poder torná-la novamente funcional.

1.1 Objetivos

Como objetivo geral, este trabalho buscar criar um ataque, com base em ferramentas anteriores [18] [36], onde um usuário no atacante possa interceptar o tráfego *web* entre uma vítima e o *gateway* da subrede, lidando com qualquer tipo de cifragem, e renderize as páginas da *web* que serão visualizadas pela vítima.

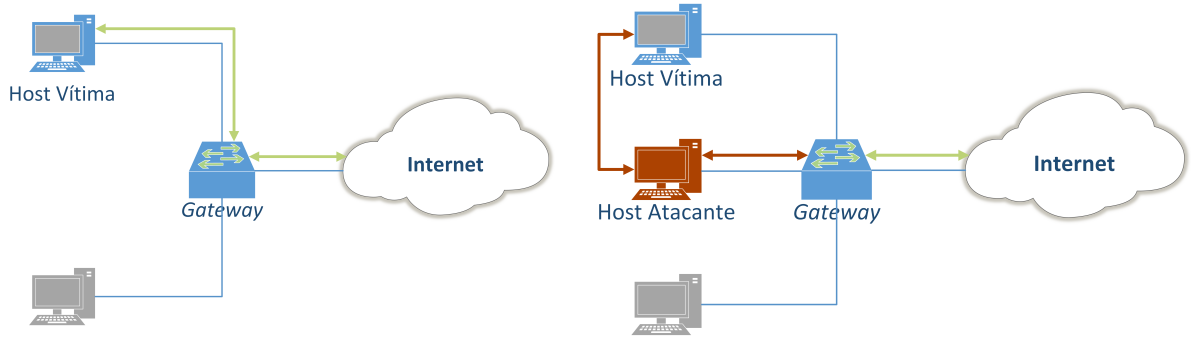
Especificamente, este objetivo abrange alguns outros específicos que podem ser traçados em traçados duas vertentes: uma teórica e outra prática. Ambas irão se complementar, e ao mesmo tempo, trazer resultados distintos.

No âmbito teórico, tem-se os seguinte objetivos:

- Entendendo as bases do ataque em questão, explicar como a técnica pode se manteve viável nas infraestruturas de redes atuais, no que diz respeito à redes locais baseadas em conexão com e sem fio (*wireless*);
- Detalhar o porquê de ser difícil defender-se do ataque;
- Entendimento do ataque como forma de criar um estudo de defesa ao mesmo.

As Figuras 1.1 e 1.2 dão um panorama do que deve ocorrer em ambos os meios de conexão. Dentro deste escopo, a aplicação criado deverá ser capaz de implementar o ataque propriamente dito.

Ressalta-se que a finalidade deste projeto é puramente acadêmica, abordando técnicas de invasão amplamente conhecidas neste âmbito. Este trabalho incentiva apenas a pesquisa sobre o tema como forma de encontrar maneiras de prevenir o mesmo. O autor e seu orientador não se responsabilizam por qualquer uso indevido ou inadvertido a partir da utilização da ferramenta e técnicas aqui abordadas, em qualquer tipo de uso fora do ambiente acadêmico. Todo o conteúdo e artefatos de software aqui criados são de guarda exclusiva do autor e seu orientador.



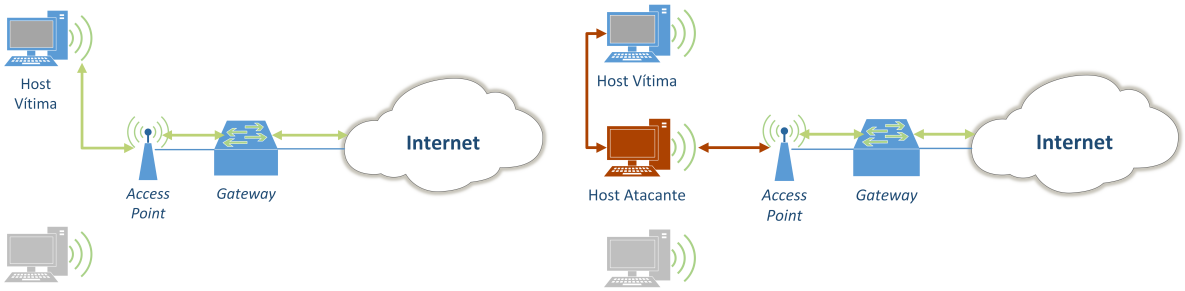
(a) Situação pré-ataque

(b) Situação pós-ataque

Figura 1.1: Estrutura do ataque em rede cabeada

1.2 Metodologia

Com o intuito de traçar estratégias eficientes, justificar decisões de projeto, detalhar problemas encontrados e suas soluções, será necessário uma abordagem teórica sobre diversos tópicos relacionados ao trabalho.



(a) Situação pré-ataque

(b) Situação pós-ataque

Figura 1.2: Estrutura do ataque em rede sem fio

Dessa forma, será necessário especificar uma subrede, seus elementos, topologias relacionadas e protocolos que regem a comunicação dentro destas e entre subredes distintas [16]. Dentro deste tópico, serão abordadas redes que utilizam o enlace sem-fio, as quais utilizam protocolos da família 802.11 [25], uma vez que este cenário encontra-se em grande uso, tanto em redes domésticas como corporativas.

O próximo passo será o entendimento sobre a interceptação de tráfego, baseado na abordagem *man-in-the-middle*. Aqui serão abordados tópicos em escuta de dados da rede (*sniffing*), falsificação de pacotes de protocolos (*spoofing*).

Seguindo, é necessário o entendimento dos detalhes essenciais para filtragem e utilização dos dados HTTP para renderização das telas. Será preciso entender como a encriptação por SSL/TLS ocorre e como evita-la. Dessa forma, esses protocolos de camadas superiores serão estudados.

Finalmente, serão mostrados alguns tópicos sobre requisições assíncronas da *web* e como estas serão abordadas pelo programa, para entender como será o suporte de renderização das telas para o atacante.

1.2.1 Escolha da Linguagem

A linguagem que foi escolhida para a aplicação foi o C++. Esta escolha foi motivada por questões de preferencia pessoal do programador e por detalhes técnicos. Além disso, a linguagem oferta bibliotecas clássicas de suporte à escuta e construção de pacotes de redes.

Adicionalmente, o C++ oferece ao programador um bom ferramental de manipulação dos dados da memória, com a utilização de ponteiros e funções inerentes aos sistemas operacionais Unix que permitem tais manipulações. Estas características, aliadas a uma programação focada em performance, dão mais robustez ao ataque.

Outra questão, foi o paradigma da linguagem. Neste trabalho, observou-se que havia a possibilidade de divisão em módulos, cada um com atribuições e responsabilidades bem definidas. Esta arquitetura não só facilita a organização e manutenção do código, como também da lógica do mesmo. Como o C++ oferece suporte à orientação a objetos, este foi outro ponto crucial.

1.2.2 Módulos da Aplicação

Por fim, a divisão do trabalho seguiu uma arquitetura simples e intuitiva. Cada entidade trabalha com entradas e saídas, além de instâncias globais. Cada módulo tem sua função bem definida e a arquitetura da aplicação seguiu algo semelhante ao mostrado na Figura 1.3:

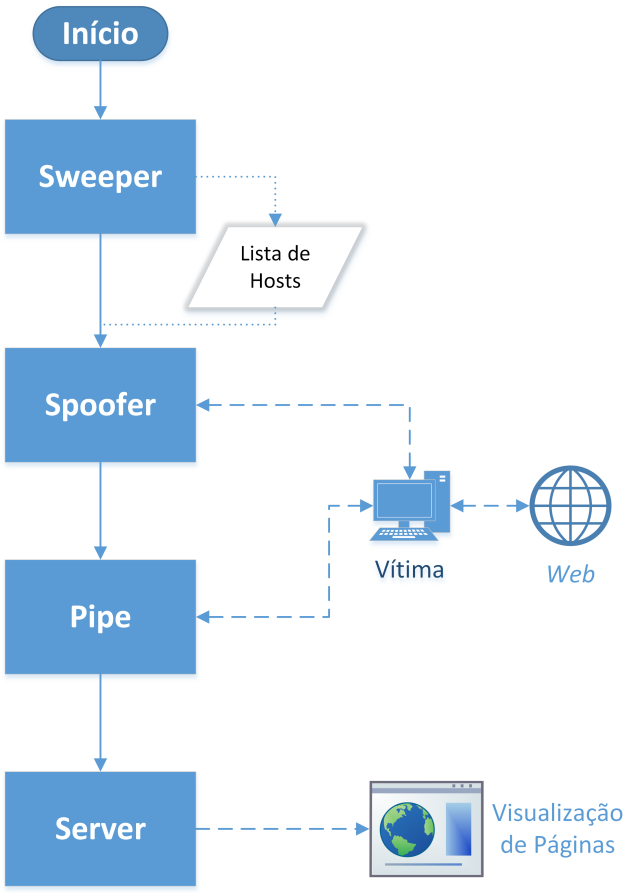


Figura 1.3: Arquitetura da aplicação com a integração dos módulos e vítima na rede

- ***Sweeper*:** módulo que irá fazer a varredura da rede, identificando quais são as máquinas ativas, disponibilizando endereços e outras informações interessantes à aplicação;
- ***Spoofer*:** módulo que implementa o ataque, falsificando pacotes ARP e enviando às vítimas. O mesmo continua enviando estes pacotes periodicamente para garantir o ataque;
- ***Pipe*:** garante uma conexão virtual entre dois *hosts*. Utilizado para que as vítimas possam se comunicar. Também estarão contidas as funções para driblar a proteção SSL;
- ***Server*:** garante a visualização de telas acessadas pela vítima. Aqui o trabalho será de receber os pacotes e renderizar as telas de acordo com as informações contidas nestas, garantindo também o tratamento às requisições *JavaScript*.

1.3 Organização do Trabalho

No Capítulo 2 será feito um estudo acerca da teoria geral em redes de computadores e redes sem fio. Serão analisados todos os aspectos de endereçamento, roteamento, segurança e demais aspectos interessantes ao projeto.

Nos Capítulos 3 e 4 serão analisadas as técnicas de ataque utilizadas, entendendo como funciona o ataque em geral, em que vulnerabilidades ele foca e outros detalhes. Já o Capítulo 5 destina-se ao entendimento da renderização das telas obtidas pelo pacote.

Por fim, o Capítulo 6 destina-se aos detalhes de implementação, e o Capítulo 7 às conclusões do trabalho.

Capítulo 2

Embasamento Teórico

Inicialmente, será definida uma rede local e as atribuições de cada dispositivo. Com isto, será especificada a pilha de camada de protocolos e, dentro desta, quais protocolos serão utilizados pelo mesmo.

2.1 Rede Local

Segundo [16] rede de computadores é uma infraestrutura envolvendo dispositivos finais, chamados *hosts* interligados por um meio de comunicação, composto por um meio propriamente dito (enlace) e dispositivos de conexão, que formam o núcleo da rede. A forma de comunicação destes é baseada em protocolos mutuamente conhecidos, os quais estabelecem padrões para troca de mensagens.

Redes são classificadas em diversas formas, além de se estruturarem de diferentes maneiras. A Tabela 2.1, definida segundo [39], cita alguns tipos de redes segundo seu alcance.

Nome	**Alcance**
`PAN`	1 até 10 metros
`LAN`	100 metros até 1km
`MAN`	5km até 100km
`WAN`	100km até 1000km
Internet	1000km até 10000km em diante

Tabela 2.1: Caracterização de redes segundo seu alcance

Neste trabalho o foco serão as LANs, ou *Local Area Networks*, que traduzem bem o contexto de redes domésticas e de pequenas corporações. Aqui, o espaço de endereçamento de máquinas na rede é limitado à esta. Assim, supondo um atacante inserido em uma LAN, a gama de vítimas disponíveis a um ataque a nível de rede limita-se aos *hosts* desta.

2.2 Camadas de Rede

Um modelo de camadas de rede abstrai as entidades de uma rede , tanto de *hardware* como de *software*, e a divide em níveis, atribuindo responsabilidades a cada um destes

níveis [35]. Ainda, cada camada deve se preocupar em comunicar-se apenas com camadas imediatamente acima ou abaixo dela.

Os dois modelos mais utilizados atualmente são o TCP/IP e o OSI. A Figura 2.1 dá uma visão melhor destes modelos.

No modelo OSI, a seguinte estrutura é definida, de baixo para cima, identificada em [42]:

- **Camada Física:** aqui tem-se a camada mais embasada em *hardware*. Ela é responsável pelo estabelecimento de padrões para transmissões de dados em sinal digital para um dado sinal físico, definindo a tradução dos *bits* por diferentes meios: elétricos, ópticos, radiofrequências, entre outros;
- **Camada de Enlace:** esta camada garante que o canal que conecta dois nós adjacentes (*hosts* ou comutadores) seja um meio de conexão confiável, seguro e livre de erros, além de assegurar o controle de fluxo e acesso a este canal, naos dois sentidos. Aqui também está o endereçamento físico dos nós e transferência entre nós de uma mesma rede;
- **Camada de Rede:** camada responsável por realizar o repasse e o roteamento de pacotes de dados entre dois *hosts*, utilizando um esquema de endereçamento lógico, permitindo a transferência de dados entre *hosts* de redes diferentes;
- **Camada de Transporte:** a função principal aqui é garantir uma transferência de dados entre dois sistemas finais, de forma completa e correta, garantindo qualidade e controle no envio dos dados, eliminando para as camadas acima as preocupações acerca de transferência de dados;
- **Camada de Sessão:** controla a sessão estabelecida entre dois processos que estão se comunicando, iniciando, gerenciando e finalizando a conexão entre estes. Aqui podem ser fornecidos serviços acerca de segurança e sincronização entre os *hosts*;
- **Camada de Apresentação:** tem a finalidade de traduzir e adaptar corretamente os dados entre dispositivos e redes diferentes. Aqui, aplica-se a conversão de caracteres, dados, estruturas de dados e codificação/decoficação de cifras, se necessário;
- **Camada de Aplicação:** camada onde funcionam as aplicações, as quais irão utilizar a rede para transferência de dados.

No modelo TCP/IP, utilizou-se a seguinte divisão em 5 camadas, definida em [16]:

- **Camada Física:** semelhante a camada física do modelo OSI. A preocupação aqui é a padronização da transmissão de *bits* por um meio físico;
- **Camada de Enlace:** assim como no modelo OSI, a camada de enlace aqui garante prover uma conexão livre de erros entre nós, a fim de interligar todos os nós de uma rede para garantir a transferência de dados entre eles. O endereçamento físico também está contido aqui, e os blocos de dados são chamados quadros;
- **Camada de Rede:** base para a *Internet*, como é conhecida. Esta camada provê a transferência de pacotes de um *host* de uma rede a outro de outra rede, possibilitando a interligação de diversos sistemas finais. Aqui o roteamento dos pacotes é garantido de forma eficiente, por meio de algoritmos de roteamento;

- **Camada de Transporte:** responsável pela conexão entre aplicações de dois sistemas finais, de modo confiável, ordenado e eficiente, com controle de fluxo e de congestionamento da rede. Os pacotes de dados transportados aqui são chamados segmentos.
- **Camada de Aplicação:** última camada do modelo, destinada à aplicação que irá utilizar a rede e protocolos próprios. Esta é geralmente implementada nos *hosts*.

Figura 2.1: Modelos predominantes de pilhas de protocolos

2.3 Protocolos

Protocolos são padrões estabelecidos para que a comunicação entre duas entidades seja feita. A definição de protocolos no âmbito de redes de computadores é largamente usada, uma vez que não só facilita o projeto de *software* e *hardware*, como também permite que dispositivos com arquiteturas e implementações distintas consigam se comunicar, possibilitando a formação de redes heterogêneas, tanto a nível de plataforma quanto de implementação [39].

2.3.1 Endereço MAC: Endereçamento Físico

Todos os nós de uma rede - sejam de borda ou de núcleo - possuem um endereço físico próprio, relativo à camada de enlace. Normalmente, o endereço utilizado é o endereço MAC (*Media Access Control*), definido detalhadamente em [24].

Este está localizado no *hardware*, sendo pertencente não à máquina, mas sim ao adaptador de rede desta. Ele é fixo, único e teoricamente imutável, independente da posição geográfica ou da rede do *host*. Adicionalmente, ele não cria uma hierarquia dentro da sua faixa de endereços, de forma que todos os nós endereçados estão em um mesmo patamar.

Este endereço possui 6 *bytes*, divididos em 6 conjuntos de 1 *byte*. Sua representação extensa se configura em 6 conjuntos com 2 dígitos hexadecimais.

Para garantir a unicidade global, a definição de um endereço é feita em duas partes: a primeira metade, comporta por 24 *bits*, é definida pelo IEEE (*Institute of Electrical and Electronics Engineers*), entidade que coordena os padrões relativos à *Internet* e seus protocolos, e a segunda metade é definida pelo fabricante.

Em uma rede, quando se deseja realizar um *broadcast* - implementação onde o pacote é mandado para todos os *hosts* da rede - utiliza-se o endereço FF:FF:FF:FF:FF:FF, que corresponde à cadeia de 48 de *bits* preenchidos com o *bit* 1.

2.3.2 Protocolo *Ethernet*

O protocolo *Ethernet* é o protocolo destinado à conexão e comunicação de *hosts* dentro de uma sub-rede, pertencente à camada de enlace, detalhadamente descrito em [24]. Tem posição dominante no mercado e pouca probabilidade de sair deste, devido à sua alta disseminação, simplicidade, baixo custo e crescente aumento na capacidade [16].

O quadro *Ethernet* possui uma variação entre um mínimo de 50 e máximo de 1500 *bytes*, excluindo os 14 *bytes* de cabeçalho. Este último representa o valor da MTU (*Maximum Transmission Unit*), adotada atualmente para transmissão de pacotes pelas LANs. A divisão explicitada na Figura 2.2 demonstra o cabeçalho *Ethernet*, que possui os seguintes campos:

- **Preâmbulo:** composto por 8 bytes, sendo o último deles diferente dos demais. Dois *hosts* dificilmente têm velocidades de transmissão compatíveis. Desta forma, estes bytes são utilizados para sincronização das taxas de transmissão dos sistemas, onde o último *byte* destacado sinaliza a chegada do quadro. Por ser este um campo de padronização de envio de dados, normalmente não é levado em conta na soma de *bytes* do cabeçalho;
- **Endereço de Destino:** com tamanho de 6 *bytes*, contém o endereço MAC do *host* destinatário;
- **Endereço de Origem:** com tamanho de 6 *bytes*, contém o endereço MAC do *host* remetente;
- **Campo de Tipos:** aqui é explicitado o protocolo de rede que o quadro carrega. Possui o tamanho de 2 *bytes*;
- **Campo de Dados:** campo variável de 46 a 1500 *bytes*, onde está o conteúdo destinado à camada de rede. O tamanho máximo vem da MTU. Caso o tamanho mínimo não seja atingido, ocorre o *stuffing*, técnica onde se colocaria *bytes* adicionais inúteis para atingir a cota campo. Estes são ignorados ao ler o cabeçalho do datagrama.
- **CRC (*Cyclical Redundancy Check*):** campo para o dígito de verificação da integridade do quadro, com detecção de erros no mesmo. Este sempre é anexado ao fim do quadro pela interface de rede, antes de ser enviado ao enlace.

Os campos que realmente configuram o cabeçalho *Ethernet* são os de endereço do rementente, o endereço do destinatário e o código do tipo, somando 14 *bytes*, sendo estes os campos interessantes neste trabalho.

Quando uma máquina envia um quadro ao enlace, outros *hosts* podem estar compartilhando do mesmo, logo, todos acabam recebendo o quadro. Entretanto, o adaptador de

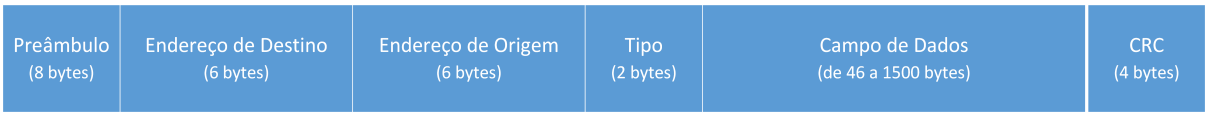


Figura 2.2: Quadro *Ethernet*, com tamanho máximo de 1526 *bytes*

rede de cada *host* checa o campo de endereço MAC de destino do cabeçalho do quadro recebido e, caso este seja o seu MAC, repassa o conteúdo do campo de dados para a camada de rede. Caso contrário, o mesmo é descartado.

Durante o translado pelo enlace, um quadro pode ter algum dado comprometido. Assim, o campo de CRC é utilizado para a checagem de integridade dos dados, onde um destinatário computa seu próprio CRC e compara com o do quadro recebido. Caso haja discrepância entre os campos, haverá um descarte do quadro. Entretanto, o destinatário não avisa o remetente sobre o descarte e, assim, o remetente assume que o quadro foi entregue corretamente, dando ao protocolo *Ethernet* um status de protocolo não-confiável [16]. A responsabilidade desta confiabilidade é repassada às camadas superiores, como será visto posteriormente.

O protocolo *Ethernet* é um protocolo de acesso múltiplo baseado no CSMA/CD. Neste, é utilizada a detecção de portadora para evitar transmissões simultâneas de dois *hosts*, garantia de detecção de colisões e controle de fluxo para retransmissões nas colisões. Assim, os adaptadores de rede devem garantir suporte à detecção de transmissões alheias e possíveis colisões.

Além disso, é importante notar que o protocolo sofreu diversas modificações ao longo dos anos. A mais notável é na sua velocidade de transmissão, que pulou de $10Mb/s$ para $1000Mb/s$. Também há diferenças entre versões do protocolo relativas ao meio que este utiliza e a capacidade de transmissão nos dois sentidos do canal.

2.3.3 Protocolo IPv4

Protocolo da camada de rede, o *Internet Protocol*, é, atualmente, o cerne da *Internet*. Este protocolo possui diversas funções, sendo as mais importantes garantir o endereçamento lógico em uma LAN e estruturar o repasse dos datagramas da camada de rede entre os *hosts*.

A junção de um cabeçalho IP, mais o corpo de dados que ele carrega, é chamada de datagrama. Um cabeçalho IP, geralmente, possui o tamanho de 20 *bytes*, relativos aos campos fixos do protocolo, definidos em [30]. Entretanto, pode conter mais dados, presentes em um campo opcional.

Os campos fixos são:

- **Versão:** com tamanho de 4 *bits*, indica qual versão do protocolo é usada: versão 4, 6 ou variantes;
- **Comprimento do cabeçalho:** possui o tamanho de 4 *bits*. Informa o comprimento total do cabeçalho, indicando em que *byte* começam os dados, uma vez que o datagrama pode conter campos adicionais;

- **Tipo de Serviço:** *byte* que determina o tipo de serviço requisitado pelo datagrama na camada de rede, por exemplo: confiabilidade, alta vazão, pouco atraso, entre outros;
- **Comprimento do datagrama:** tamanho de 2 *bytes*, informa o comprimento total do datagrama, incluindo cabeçalho e dados, em *bytes*;
- **Identificador de pacote:** 2 *bytes* de grande importância do cabeçalho. Usado na fragmentação dos datagramas IP quando é necessária tal técnica, para futura remontagem ordenada do pacote;
- ***Flags:*** 3 *flags* de 1 *bit* cada, também usadas no processo de fragmentação;
- **Deslocamento de Fragmentação:** composto por 13 *bits*, também é usado na fragmentação de datagramas. Indica o número do *byte* esperado para o próximo datagrama;
- **Tempo de Vida:** composto por 1 *byte*, existe para que um datagrama não vague indefinidamente na rede. A cada passagem por um roteador, este contador é decrementado e, ao ser zerado, o datagrama é descartado;
- **Protocolo da camada superior:** campo de 1 *byte* usado para indicar ao destino final do datagrama qual o protocolo de camada de transporte utilizado para o envio, para que o destinatário se alinhe com o mesmo;
- **Soma de verificação:** tamanho de 2 *bytes*, é utilizado por cada roteador onde o datagrama passa, para a checagem da integridade dos dados;
- **Endereço de Fonte:** endereço IP de 4 *bytes* do *host* remetente;
- **Endereço de Destino:** endereço IP de 4 *bytes* do *host* de destino;
- **Opcionais:** local onde é possível adicionar campos opcionais ao cabeçalho do datagrama. O objetivo aqui é utilizar este campo apenas quando necessário, para informações adicionais cruciais.

Assim como o *Ethernet*, o IP também possui uma MTU de 65536. Entretanto, esta MTU é muito superior à MTU do enlace, uma vez que o limite de 1500 *bytes* deve ser respeitado. Dessa forma, um esquema de fragmentação é introduzido no IP. Entretanto, este é transparente para certas aplicações, sendo o Webspy uma delas. O protocolo IP possui diversas outras peculiaridades, porém serão abordados apenas os quesitos de endereçamento e sub-redes, os quais são focos deste trabalho.

Endereço IP

O endereço IP é composto por 4 bytes, onde em notação humana é escrito na forma de 4 números de 0 a 255, separados por ponto. O endereço 192.168.1.147, por exemplo, seria escrito na forma binária 1000000 10101000 00000001 10010011. Este endereço é lógicamente dividido em duas partes: uma para definir o endereçamento de uma sub-rede e a outra para definir o endereçamento de um *host* na sub-rede da primeira parte, definido e especificado em [10].

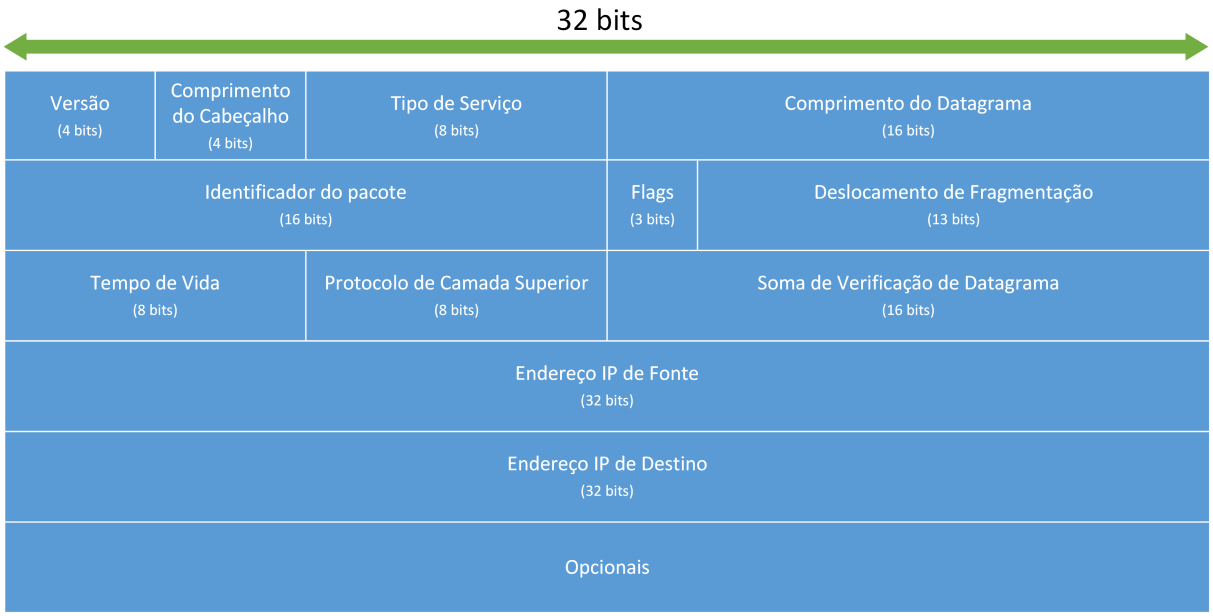


Figura 2.3: Cabeçalho do datagrama IP

A identificação da sub-rede é criada a partir de uma máscara de rede. Esta é uma abstração que utiliza uma faixa de *bytes* do endereço para identificação da sub-rede. Classes de endereço IP se referem à porções pré-estabelecidas destes *bytes* para referênciar sub-redes, descritas na Tabela 2.2. O restante dos *bytes* é utilizado para endereçamento dos *hosts*. É interessante notar que, quanto maior a máscara da rede, menor a quantidade de endereços que esta dispõe para os *hosts*.

Classe	Máscara	*Hosts*
A	255.0.0.0	16.777.214
B	255.255.0.0	65.534
C	255.255.255.0	254
D	-	*Multicast*

Tabela 2.2: Classes de endereços IP

Endereçamento de *Hosts*

Ao ser utilizada, cada interface deve ganhar um endereço IP para ser localizada na rede. Desta forma, uma mesma máquina pode ter vários endereços, um para cada interface. Isto é bem comum em computadores convencionais que trazem uma interface para *Ethernet* e outra para *wireless*. Roteadores também possuem várias interfaces diferentes, mas estas têm a finalidade de segmentar redes logicamente, criando divisões da rede, ou sub-redes.

Finalmente, pode-se definir uma sub-rede como: divisão de uma rede onde todos os nós têm endereços IP com a mesma sub-máscara de rede. Sub-redes configuram um segmento de comunicação comum à vários *hosts* e, a troca de dados com algum outro *host* de uma sub-rede distinta, implica no repasse de pacotes efetuado por um roteador ou comutador que lide com camada 3.

A Figura 2.4 mostra como as sub-redes se dividem em torno de um roteador. É importante observar que não é necessário que exista um *host* na sub-rede para que ela exista e que, elementos de núcleo como roteadores, também são considerados.

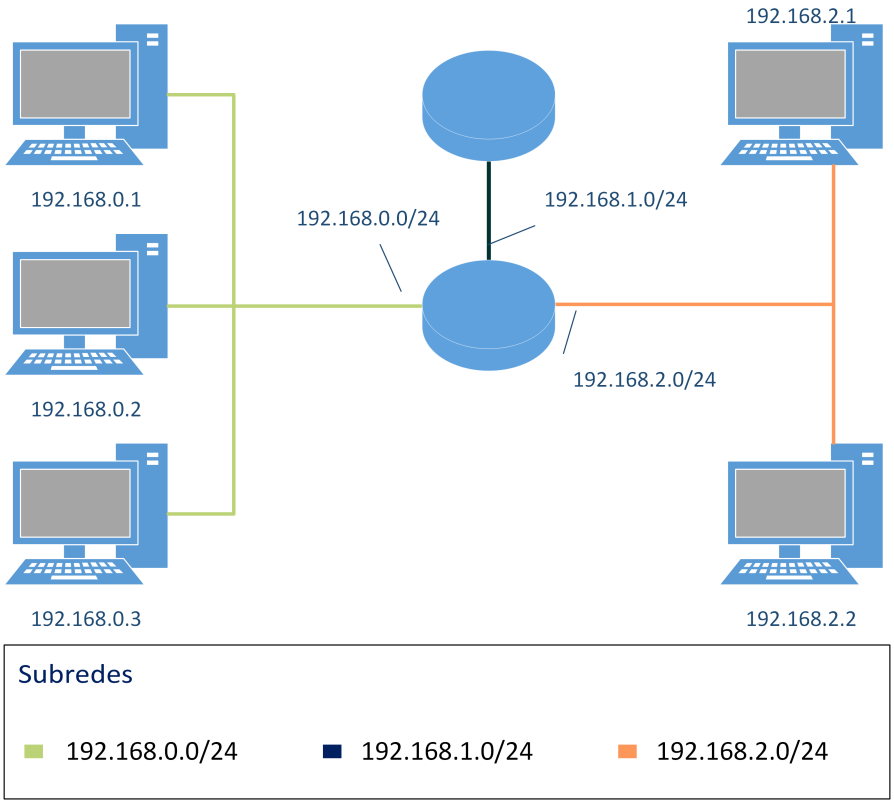


Figura 2.4: Exemplo de uma rede composta por 3 sub-redes

Dessa forma, percebe-se como uma sub-rede é estruturada e quem faz parte desta: todos os nós que estão ligados por equipamentos de camada 1 e 2 apenas, ou seja, sem divisão de rede, e que possuem a mesma máscara de sub-rede. Assim, em termos do ataque aqui estudado, todos os *hosts* deste limite são possíveis vítimas.

2.3.4 Associação de Endereços: Protocolo ARP

Ao citar o endereçamento de rede, dois tipos são foco: o físico (MAC) e o lógico (IP). Em uma rede, ambos são importantes no processo de identificação do *host*, uma vez que a comunicação entre dois *hosts* necessita de ambos os níveis. Quando um nó deseja enviar dados a outro, os pacotes são preenchidos com o IP e o MAC de destino, porém, é necessário ter conhecimento desta dupla de endereços. Como um *host* entende qual MAC pertence a um dado IP?

O protocolo ARP (Address Resolution Protocol) cuida desta tarefa, trabalhando com um esquema de associação de endereços MAC e IP, guardando esta informação em uma *cache* local de cada *host*. É um protocolo da camada de enlace, sendo essencial para a comunicação entre nós de uma sub-rede.

O ARP é detalhadamente definido em [29] e possui diversas operações. Seu cabeçalho tem tamanho e campos fixos, como mostrado na Figura 2.5, onde não há mais além dos

descritos. A diferença para cada operação está na forma de preenchimento de cada campo, sendo estes:

- **Operação *Ethernet*:** campo destinado apenas à identificação do protocolo de enlace em vigor, normalmente sendo este o Ethernet, indicado pelo código `0x01`;
- **Protocolo alvo:** campo destinado a indicar o tipo do protocolo lógico com o qual o ARP está lidando. Normalmente é utilizado o protocolo IP, indicado pelo código `0x0800`;
- **Tamanho do endereço físico:** tamanho em bytes dos endereços físicos a serem tratados. No caso do *Ethernet* são utilizados 6 bytes (`0x06`);
- **Tamanho do endereço lógico:** tamanho em bytes dos endereços lógicos a serem tratados. No caso do IP são utilizados 4 bytes (`0x04`);
- **Código da Operação:** código da operação ARP a ser feita. Este campo determina como o *host* que receber o pacote ARP irá agir em seguida;
- **MAC do remetente:** endereço MAC do *host* que envia o pacote ARP;
- **IP do remetente:** endereço IP do *host* que envia o pacote ARP;
- **MAC de destino:** endereço MAC do *host* que recebe o pacote ARP;
- **IP de destino:** endereço IP do *host* que recebe o pacote ARP.

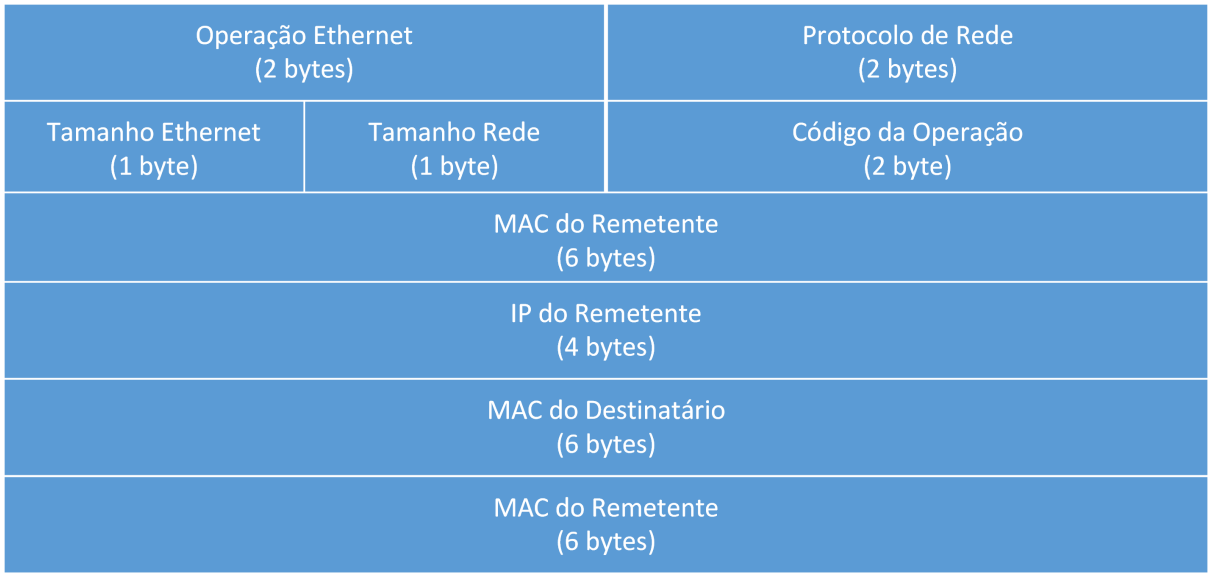


Figura 2.5: Cabeçalho ARP

Para realizar as associações de endereços, o protocolo ARP dispõe de duas mensagens, uma de pergunta e outra de resposta. Para entendê-las ambas são explicadas na ordem lógica do protocolo:

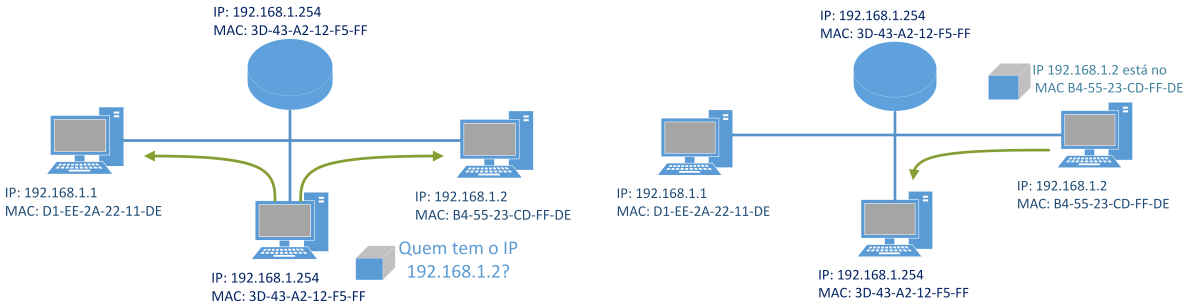
- **ARP *Request*:** é uma mensagem enviada em *broadcast* (para todos da sub-rede) onde é requisitado o MAC associado a um dado IP. O cabeçalho é preenchido com o MAC e o IP do requisitante e, nos campos destinados ao destinatário, são inseridos o IP, cujo endereço MAC deseja-se saber, e no campo de MAC, uma sequência de *bytes* zerados. Todos os nós da sub-rede irão receber este pacote;
- **ARP *Reply*:** resposta à mensagem anterior, enviada em *unicast* ao requisitante. O campo que continha o MAC nulo é preenchido com o MAC do *host* correspondente, que possui o IP requisitado. A mensagem, entretanto, retorna com um código diferente no campo de operação, indicando uma resposta.

Com estas duas operações, uma máquina pode descobrir a quem pertence qualquer MAC de uma faixa de IPs desejada. O esquema da Figura 2.6 mostra como funciona detalhadamente a troca de mensagens.

Para não necessitar de uma consulta de endereço para cada quadro enviado, cada *host* possui uma *cache* local, denominada Tabela ARP, que irá guardar cada endereço IP, seu MAC correspondente e o horário de associação, para implementação de um *timeout*, evitando informações defasadas. Logo, quando o *host* deseja enviar um datagrama para um nó, ele irá obter o endereço MAC procurando nesta tabela pelo IP correspondente, caso não o encontre, irá lançar mão da ARP *Query*.

Além disso, é comum um nó aproveitar uma ARP *Request* para preencher sua Tabela ARP, uma vez que, quando um *host* A deseja comunicar-se com um *host* B, é esperado que o contrário também aconteça. Isto diminui a quantidade de *broadcasts* na rede.

Entretanto, a tabela ARP possui duas características interessantes: ela não guarda estado e seu preenchimento é totalmente passivo. A primeira característica significa que uma entrada na tabela pode ser atualizada de modo livre, sem checagem de entradas anteriores. Logo, um IP pode mudar de MAC diversas vezes na *cache* de um *host*. A segunda característica mostra que não é necessário que um nó requisite uma resposta ARP para atualizar sua tabela, basta receber uma ARP *Request* e a tabela será atualizada. Este comportamento é intrínseco ao protocolo, para diminuir seu *overhead* na rede, onde um *host* sempre aproveita uma informação ao recebê-la.



(a) Esquema de ARP *query*

(b) Esquema de ARP *response*

Figura 2.6: ARP *query* em sub-rede com 3 máquinas

2.4 Hardware de Rede

Especificado o funcionamento lógico de uma sub-rede e como funciona a comunicação e localização de *hosts*, é necessário entender os tipos de dispositivos que serão utilizados, os quais participaram do ambiente de ataque.

Inicialmente, o ambiente estava limitado à redes cabeadas, em ambiente físico paupável. Porém, com o advento da tecnologia sem fio, predominante no mercado atual, um novo ambiente foi criado. Neste capítulo, será dada uma base teórica dos dispositivos do ambiente cabeado e, na Seção 2.6, será mostrado o ambiente *wireless*, em seus aspectos lógicos e físicos.

2.4.1 Conceitos e Propriedades

Como foi dito, cada *host* possui, pelo menos, um adaptador de rede. Mais especificamente, a NIC (*Network Interface Card*) é o *hardware* que irá dar o suporte aos protocolos de camada física e de enlace, além de conter o endereço MAC. Esta placa irá realizar o repasse de quadros, realizando a checagem dos mesmos e anexando o CRC ao fim do pacote.

Ao enviar um quadro, um *host* tem um destino, que pode ser alcançado diretamente pelo meio, podendo o quadro passar por um repetidor ou ser repassado por um comutador de camada de enlace, ou rede, para alcançar o nó destino. Repetidores e comutadores são dispositivos de rede, os quais permitem a conexão e transmissão de dados entre os nós. Na classe dos repetidores destacam-se os *hubs*, enquanto na dos comutadores estão os *switches* e roteadores, sendo que estes últimos atua na camada de rede, ou seja, a nível *software*.

Esses dispositivos possuem propriedades interessantes. Primeiramente, uma propriedade intrínseca dos mesmo é a regeneração do sinal recebido. Uma vez que o sinal de dados - tanto elétrico, ótico ou por radiofrequência - é passível de perder energia para o meio, por vezes é necessário o reforço do mesmo. Caso contrário, o sinal pode conter muito ruído ou estar fraco e não ser compreendido pelo receptor.

Ainda, é intrínseco a alguns comutadores aprender automaticamente os endereços alcançaveis por suas interfaces no momento en que estes são conectados à rede, ou seja, os dispositivos se auto configuram, adaptando-se à rede, dispensando uma abordagem manual. Esta propriedade é chamada de *plug-and-play*.

Por fim, segue abaixo as vantagens que os comutadores trazem à rede:

- **Eliminação de colisões:** ocorre à medida que são eliminados enlaces compartilhados por vários *hosts*. Como cada *host* é ligado à uma interface do comutador, seu tráfego é isolado e gerênciado pelo mesmo.
- **Enlaces heterogêneos:** comutadores podem atuar como pontes entre enlaces distintos, como, por exemplo, ambientes cabeados e ambientes sem fio, possibilitando redes heterogêneas;
- **Gerência:** os comutadores agem como centrais, recebendo diversos pacotes e guardando informações relevantes da rede.

2.4.2 Tipos

Serão abordados três tipos de comutadores: *hubs*, *switches* e roteadores. Importante notar que eles não atuam na mesma camada, pois *hubs* e *switches* atuam na camada de enlace e roteadores atuam na camada de rede. Alguns *switches* conseguem atuar em camadas superiores, porém, aqui, eles serão tratados como roteadores.

Hubs

Os *hubs* são dispositivos focados em reforçar e repetir um sinal. Por definição, *hubs* não roteiam dados, ou seja, não direcionam os dados ao seu destino correto. Seu funcionamento é dado por *broadcast*: ao receber um dado por uma porta, o *hub* irá reenviar o mesmo dado para todas as outras portas. O máximo que pode ocorrer é um controle de congestionamento, onde um *hub* que esteja ocupado realizando transmissões, envia solicitações de espera à máquinas que estejam tentando comunicar-se com ele. Logicamente, esta mensagem é enviada em *broadcast*, mas endereçada para à máquina solicitante. Estes dispositivos, por não necessitarem de gerência de *hosts* e controle de dados, são dispositivos *plug-and-play*.

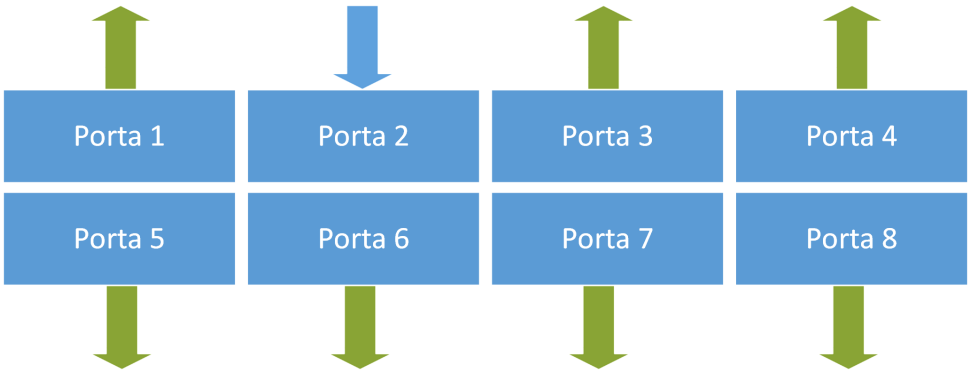


Figura 2.7: Exemplo do funcionamento de um *hub*, com um dado entrando por uma porta e sendo distribuído pelas demais

Switches

São dispositivos que possuem interfaces e realizam repasse, filtragem ou ambos dentro de uma rede, como definido em [11]. Interface é o meio de comunicação entre dispositivo e enlace, o qual será frequentemente abordado. Filtragem é a capacidade de determinar se um quadro deve ser repassado à uma interface ou descartado, enquanto o repasse é a ação de definir à qual interface um quadro deve ser enviado.

Switches utilizam uma estrutura chamada tabela de comutação, a qual mapeia quais endereços MAC estão acessíveis por suas interfaces, armazenando o instante deste mapeamento, para evitar informações defasadas. Com isto, ao receber um quadro endereçado a um determinado *host*, um *switch* repassa este quadro ao destino correto, lendo o cabeçalho do quadro. A Tabela 2.3 e a Figura 2.8 exemplificam um caso de uso do *switch*.

Aqui, percebe-se duas propriedades cruciais dos *switches*: repasse e filtragem. O repasse é o ato de determinar para qual interface um quadro deve ser direcionado e

Endereço	Interface	Horário
2B-54-C3-D8-AA-F2	8	19h28
44-9E-F8-D1-A2-EE	4	19h30
55-12-DD-C3-D5-AA	7	19h33

Tabela 2.3: Tabela de comutação hipotética do exemplo da Figura 2.8

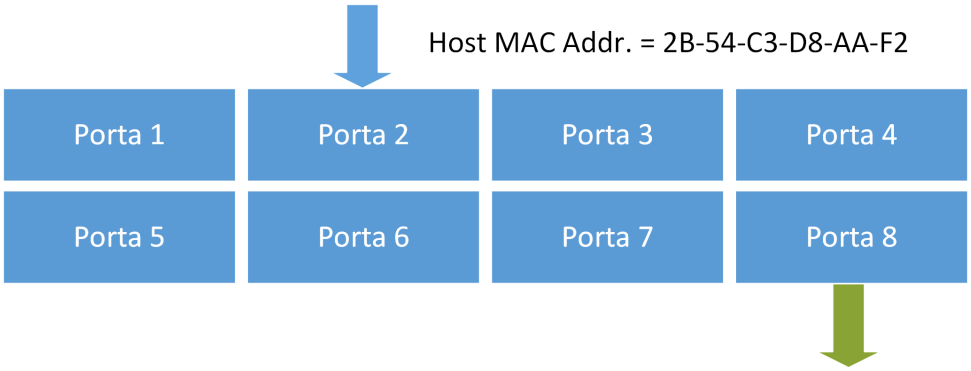


Figura 2.8: Exemplo de *switch* recebendo um dado destinado a uma máquina e fazendo o correto repasse

enviar o mesmo, enquanto que a filtragem é a capacidade de decidir se o quadro deve ser repassado ou descartado em uma dada situação. A partir daí, o *switch* consegue garantir o isolamento de tráfego de dados, enviando o quadro para o segmento de rede correto, evitando o *overflow* de pacotes.

Roteadores

Por fim, tem-se os roteadores. Estes são dispositivos mais poderosos. O esquema de checagem de endereços baseia-se nos endereços do protocolo IP e a rota tomada pelos dados é predeterminada por algoritmos de roteamento especializados. Assim, garante-se um roteamento eficaz dos dados.

A estruturação de um roteador é dada pelas portas de entrada dos pacotes, que são as interfaces conectadas aos enlaces; por um núcleo contendo a elemento de comutação, onde é feito o direcionamento dos pacotes vindos das portas de entrada para as corretas portas de saída e pelas portas de saída por onde os pacotes serão entregues aos enlaces. As portas possuem *buffers* que podem guardar pacotes caso a taxa de chegada seja mais alta que a taxa de saída.

Estas características obrigam que a estrutura de um roteador seja bem mais complexa e robusta do que a de um *switch*, tanto no nível de *software*, onde é necessário lidar com áreas de memória, processamento dos algoritmos de roteamento e gerência do dispositivo, quanto no nível de *hardware*, com controle de fluxo nas portas de entrada, matriz de comutação eficiente e das interfaces. Entretanto, o funcionamento dos roteadores não é o foco neste trabalho, mas sim a sua função no envio correto dos dados em uma rede.

Assim, basta saber que, roteadores são responsáveis pelo isolamento do tráfego na camada de rede e também isolam sub-redes, ou seja, cada interface de um roteador está

conectada a uma sub-rede diferente, com uma sub-máscara diferente. Desta forma, o ataque estará associado à uma destas sub-redes.

Serviço	**Hub**	***Switches***	**Roteadores**
Isolamento de Tráfego	Não	Sim	Sim
Plug-and-play	Sim	Sim	Não
Roteamento Ótimo	Não	Não	Sim

Tabela 2.4: Comparativo entre os tipos de dispositivos de núcleo em uma rede

2.5 Localização de Vítimas

Esta seção é uma síntese de tópicos abordados anteriormente com foco em uma das funcionalidades da ferramenta: a localização de *hosts* classificados como possíveis vítimas. Como é levado em conta que o atacante já terá ingressado na rede local, ou seja, com um IP pertencente ao daquela rede, este poderá ser facilmente localizado na mesma. Além disso, o *host* terá acesso ao endereço do *gateway*, podendo conectar-se à diferentes redes e, consequentemente, à Internet.

Neste trabalho será utilizado o exemplo de rede local mostrado na Figura 2.9. Note que esta possui dois enlaces diferentes: um com fio e outro sem fio. Além disso, esta possui um novo *hardware* de rede: um *access point*. Este será abordado em capítulos posteriores, porém, basta saber que poderá agir como o *gateway* da subrede em que está inserido.

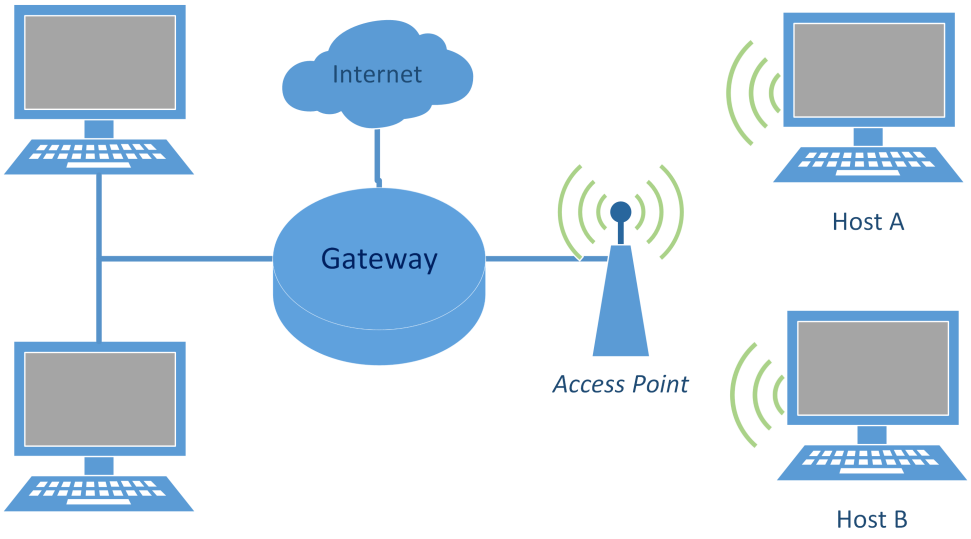


Figura 2.9: LAN heterogênea que será utilizada como referência para o trabalho

2.5.1 *Sweep* em Rede

Para o funcionamento ferramenta Webspy será necessário um módulo de detecção de vítimas na sub-rede. Este módulo deverá fazer uma varredura pela mesma, de modo que

irá construir uma lista de vítimas, com seus respectivos endereço IP e endereço MAC. Com estas informações, o atacante poderá enviar pacotes para as vítimas, dando margem ao ataque. Esta varredura pode ser feita de dois modos, já presentes na primeira versão da ferramenta [18]: ARP *sweep* ou *ping sweep*.

O *ARP sweep* é a técnica de varredura baseada no protocolo ARP. Aqui, a descoberta dos *hosts* e seus endereços é feita baseada no esquema de ARP *requests*. A técnica é baseada na iteração por uma faixa de endereços IP, onde a cada iteração é feita uma ARP *query* direcionada ao IP em questão. Caso haja resposta, o *host* está ativo e é registrado com suas informações, como possível vítima. Caso não haja resposta em um dado tempo, o endereço é descartado e o nó é considerado como não ativo.

O *Ping Sweep* é implementado usando o `ping`, utilitário baseado no protocolo ICMP, testando a conexão com um dado *host* e trazendo suas informações. Por meio do `ping`, é realizada uma iteração sobre uma dada faixa de IPs, disparando requisições do tipo *echo request*. Caso o *host* com o determinado IP exista, ele responde usando a diretiva *echo reply*, informando seus dados e indicando sua atividade.

Observando as características de ambas as técnicas, é possível fazer uma comparação entre as duas. Por usar uma ARP *query*, o ARP *sweep* irá utilizar o *broadcast* a cada checagem de endereço, o que gera muito tráfego na rede, aumentando as chances de detecção do ataque. Já o *ping sweep* gera mensagens em *unicast*, o que o torna mais discreto. Além disso, o *ping sweep* pode alcançar *hosts* de outras sub-redes, uma vez que atua em camada 3, quebrando a barreira imposta pelo roteador.

Entretanto, existem padrões de segurança seguidos para sub-redes e, um deles, é a negação do envio de pacotes do tipo *echo reply* por parte de diversos dispositivos. Isto acaba por invalidar a técnica de *ping sweep*. Além disso, o ARP é um protocolo essencial para o funcionamento das sub-redes, sendo o seu tráfego considerado normal em diversas delas, mesmo que em demasia. A negação de pacotes ARP em redes por *firewalls* e outros utilitários, invalidaria o mapeamento dos nós da sub-rede, bloqueando a comunicação. Dessa forma, o ARP *sweep* é um tipo de varredura transparente para vários equipamentos de segurança e será a abordagem adotada.

2.6 Protocolo 802.11: transporte para o *Wireless*

Como abordado na introdução deste trabalho, é importante certificar-se que a ferramenta Webspy terá sucesso em ataques situados em ambiente sem fio. Este, entretanto, trás uma diferente topologia de rede, dipositivos, protocolos e detalhes de implementação, aos quais será necessário um estudo, a fim de saber se alguma solução terá de ser adotada pela aplicação.

Redes estruturadas sobre o meio sem fio são implementadas sobre o protocolo 802.11, detalhadamente definido em [25], [12] e [16], sendo estas referências a base para esta seção. O protocolo é baseado na tecnologia *Wi-Fi*, marca registrada pela *Wi-Fi Alliance*. O protocolo age sobre duas camadas: a física, apelidando-a de PHY e a de enlace, apelidando-a de MAC. Esta abstração do protocolo pode ser observada na Figura 2.10.

Aproveiando a Figura 2.10, observa-se que o protocolo possui distribuições. Estas surgiram ao longo dos anos, como formas de atualizar e melhorar o protocolo, publicadas como emendas a especificação original do protocolo. Atualmente, as distribuições mais utilizadas são a *a*, *b*, *g* e *n*. Estas distribuições trabalham com algumas características

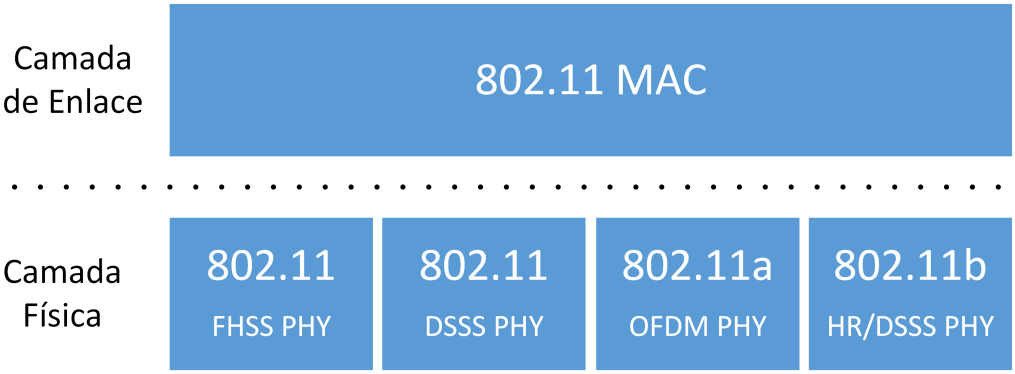


Figura 2.10: Extensão do protocolo 802.11 através das duas primeiras camadas de rede. No nível PHY é mostrada a divisão de acordo com as distribuições

físicas diferentes, culminando na não interoperabilidade entre alguns dos padrões. Consequentemente, alguns dispositivos suporta mais de uma dristribuição. A Tabela 2.5, adaptada de [12], traz um comparativo das distribuições segundo alguns critérios.

Distribuição	Faixa (GHz)	Canal (MHz)	Modulação	Taxa Máxima
a	5.15 - 5.85	22	OFDM	54Mbps
b	2.4 - 2.585	20	DSSS	11Mbps
g	2.4 - 2.585	20	OFDM	54Mbps
n	2.4 - 2.585 / 5.15-5.8	20/40	MIMO/OFDM	600Mbps

Tabela 2.5: Tabela comparativa dos padrões 802.11

2.6.1 Entidades e Topologias

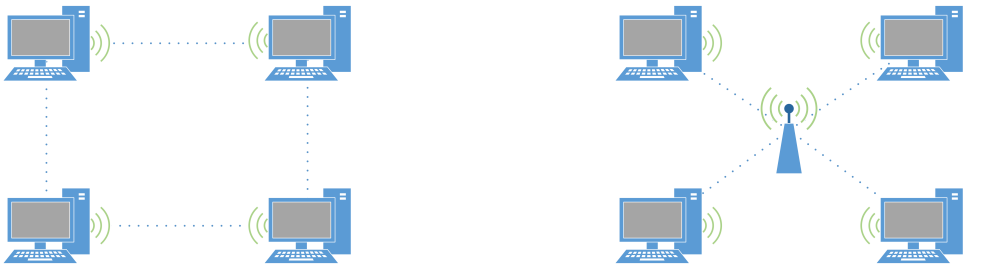
Com a introdução de um novo enlace, novas entidades são introduzidas, sendo que estas atuam diretamente sobre a abstração PHY. Estas são:

- **Estações:** analogamente as redes cabeadas, são as máquinas com conexão a rede que possuem um cartão de interface (NIC) wireless. Na literatura, são abreviadas como STAs;
- **Enlace sem fio:** é o meio de conexão entre os nós sem fio da rede. Dessa forma, este enlace é uma abstração virtual, uma vez que a conexão Wi-Fi é feita por ondas de radio-frequência, transmitidas pelo meio;
- **Meio de distribuição:** se refere ao *backbone* utilizado pela redes sem fio para o tráfego de dados entre estaoes bases, meio de enlace por fio;
- **Estação Base:** parte essencial da infraestrutura da rede sem fio, a estação base é o meio de interface entre a rede cabeada e a rede sem fio. Basicamente, é este dispositivo que está conectado ao *backbone* da *Internet* pelo meio cabeado, encapsulando os quadros *Ethernet* em quadros 802.11 para as demais stations, provendo a comunicação destas com o núcleo da rede. Exemplos destes dispositivos são as

antenas de telefonia de celular e os pontos de acessos, normalmente chamados, na literatura, de *Access Point* (AP).

Com base nestas entidades, uma rede sem fio pode se basear em duas topologias [12]:

- ***Ad-hoc*:** sem uma gerência central, as estações comunicam-se diretamente entre si, formando uma rede. Todo roteamento de informações e controle de mensagens é feita pelos nós. Pode ser observada na Figura 2.11a;
- **Infraestrutura:** assemelha-se muito a topologia de estrela de rede cabeadas, onde o AP é entidade central que irá conectar todas as estações, provendo a gerência na comunicação dos nós. Pode ser observada na Figura 2.11b.



(a) Topologia *Ad-hoc* (b) Topologia de Infraestrutura

Figura 2.11: Topologias de redes sem fio

2.6.2 WLANs

Uma WLAN (*Wireless Local Area Network*), é definida por um conjunto de uma ou várias BSSs (*Basic Service Sets*) [12]. Cada BSS é composta por um AP, utilizado uma estação de base central, um possível conjunto de estações de base subordinadas ao principal e um conjunto de estações (*hosts*)associadas. Os APs serão os responsáveis por conectarem as estações com o *backbone* da Internet ou uma LAN, normalmente estando conectadas pelo meio cabeado com estas. A Figura 2.12 representa um esquema de WLAN com 2 BSSs.

Cada BSS é identificada por um identificador, chamado SSID (*Service Set Identifier*), composto por uma cifra de 32 *bytes* quaisquer. Logo, uma BSS se anuncia por meio de pacotes de sinalização, chamados *beacons*, que carregam o SSID, emitidos pelos APs.

As estações que recebem esses pacotes, ganham conhecimento da BSS e podem requisitar o ingresso à esta. Uma estação cliente também pode lançar pacotes de buscar por BSSs, que serão respondidos pelos APs correspondentes.

O SSID não é único, de modo que duas BSSs podem ter o mesmo SSID. Entretanto, a distinção e escolha da BSS correta fica a cargo das aplicações embutidas no cliente.

O ingresso da estação á BSS pode estar sujeito a um protocolo de autenticação e cifragem de dados, para proteção da conexão e da WLAN.

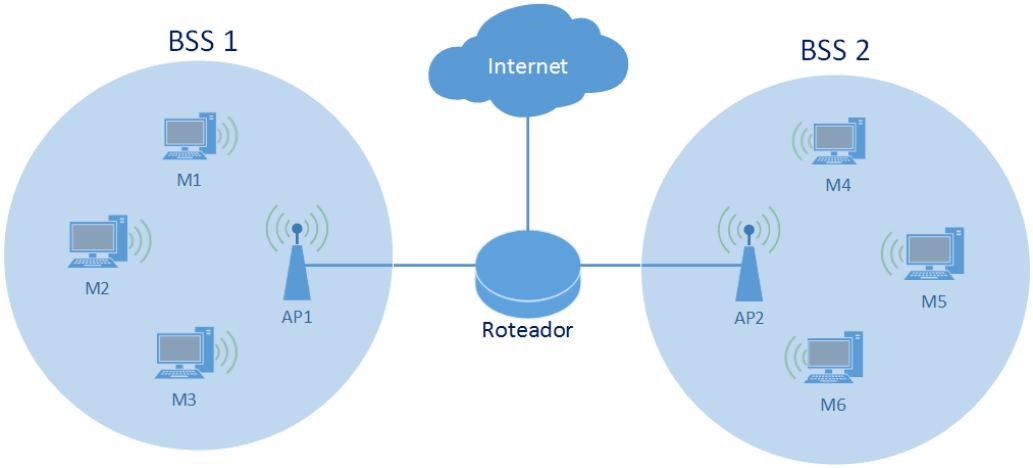


Figura 2.12: Exemplo de rede *wireless* com 2 BSSs

2.6.3 Estrutura do Quadro

O quadro utilizado na abordagem sem fio é semelhante ao quadro *Ethernet*, porém adicionado de campos específicos, como definido em [25]. A Figura 2.13 mostra a estrutura geral do quadro.

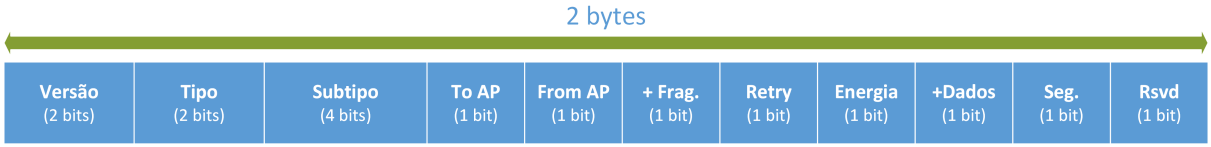


Figura 2.13: Cabeçalho do quadro 802.11

Controle de Quadro

O primeiro campo, destinado ao controle do quadro é composto pelos seguintes campos, sendo ilustrado na Figura 2.14:

- **Versão:** especifica a versão do protocolo. Como o protocolo não sofreu variações significativas, permanece o mesmo;
- **Tipo e Subtipo:** são usados para identificar a função do quadro. O tipo varia entre: controle, de dados ou gerência. O subtipo varia entre cada subconjunto de tipos destes 3 tipos principais;
- ***To AP* e *From AP*:** juntos formam uma entre quatro combinações que informam o tipo do envio: se o remetente e/ou destino são AP ou não. Estes campos informam como o protocolo vai preencher e utilizar os campos de endereço do quadro;
- **Mais fragmentos:** sinaliza se há mais fragmentos do quadro a ser enviado ou se é o último fragmento;
- **Nova tentativa:** informa se o quadro transmitido é uma retransmissão de um quadro previamente enviado;

- **Gerenciamento de energia:** possui um 1 *bit* e informa o gerenciamento de energia de uma *station*;
- **Mais dados:** informa a uma *station* que há mais dados a serem enviados para ela no *buffer* do AP;
- **Proteção do quadro:** indica se o quadro foi contém informação encapsulada por um algoritmo criptográfico;
- ***Rsvd*:** usado para sinalizar uso de classes adicionais de serviços.

Figura 2.14: Estrutura do campo de controle do quadro 802.11

Duração

O campo de duração é utilizado para indicar o tempo que um dado canal foi reservado por uma estação transmissora.

Campos de Endereço

Diferentemente do protocolo *Ethernet*, o quadro 802.11 possui 4 campos para endereços MAC, com seus significados sendo modificados a partir dos *bits To AP* e *From AP*. Os 4 podem assumir os seguintes significados:

- ***BSS Indentifier*:** identifica uma BSS. Quando o quadro é enviado por uma estação que opera no modo de infraestrutura, o o endereço é o endereço MAC do AP. Quando o modo é o *ad hoc*, o campo é preenchico com um endereço randômico a ser gerenciado pela origem;
- ***Destination Address*:** endereço MAC da estação final que deve receber o quadro. Equilavente ao MAC de destino do protocolo *Ethernet*;
- ***Source Address*:** endereço MAC da estação que criou e enviou o quadro. Equilavente ao MAC de origem do protocolo *Ethernet*;
- ***Receiver Address*:** endereço MAC da próxima estação a receber ao quadro, ou seja, a que irá fazer o repasse do quadro;
- ***Transmitter Address*:** endereço MAC da estação que repassou o quadro. Esta abstração é utilizada apenas em redes *ad hoc*.

A Tabela 2.6 serve como referência para o preenchimento dos campos de endereço do quadro segundo as *flags* indicadas.

To AP	From AP	Endereço 1	Endereço 2	Endereço 3	Endereço 4
0	0	DA	SA	BSS	-
0	1	DA	BSS	SA	-
1	0	BSS	SA	DA	-
1	1	RA	TA	DA	SA

Tabela 2.6: Referência de preenchimentos dos campos de endereço do quadro 802.11 segundos as *flags* de controle

Sequência e CRC

O campo de controle de sequência é utilizado para identificar o número sequencial do quadro, auxiliando no controle de retransmissões e ordenamento nas transmissões dos quadros.

Assim como no protocolo *Ethernet*, o protocolo 802.11 também implementa a verificação de quadros a partir de um campo CRC anexado ao fim do quadro.

2.7 Envio de um Quadro

Seguindo a lógica ilustrada na Tabela 2.6 e definida por [25], será traçado o caminho de um quadro 802.11 enviado de um *host* a outro, a fim de entender como o envio deste quadro difere do envio do quadro *Ethernet* e que impactos isto traz as aplicações.

Duas situações serão ilustradas: quando uma estação envia um quadro a outra estação e quando uma estação envia um quadro ao *gateway*, uma vez que estas são as duas abordagens com que a ferramenta Webspy irá trabalhar. A Figura 2.9, que representa a sub-rede de referência para o trabalho será utilizada para dar base a explicação.

2.7.1 Estação-Estação

Quando o *Host* A deseja enviar um datagrama ao *Host* B, o mesmo irá encapsular o datagrama em uma quadro 802.11. Neste, o *bit To AP* será definido como 1, uma vez que para alcançar o *Host* B, o *Host* A deve enviar o quadro primeiramente ao AP. Já o *bit From AP* será definido como 0. Condizente a tal, o preenchimento dos endereços no quadro será o seguinte:

- **Endereço 1:** abstraído como **BSS Identifier**, uma vez que o AP deverá receber o quadro para ser retransmitido, conterá o MAC do AP;
- **Endereço 2:** abstraído como **Sender Address**, uma vez que o *Host* A envia o quadro inicialmente, conterá o MAC do *Host A*
- **Endereço 3:** abstraído como **Destination Address**, uma vez que o *Host* B deverá receber o conteúdo do quadro, conterá o MAC do *Host* B.

Quando o AP receber o quadro, irá transmiti-lo para o *Host* B. Entretanto, agora apenas o *bit From AP* é definido como 1 e os os campos de endereços são preenchidos da seguinte forma:

- **Endereço 1:** abstraído como **Destination Address**, uma vez que o *Host* B irá receber o quadro, conterá o MAC do *Host* B;
- **Endereço 2:** abstraído como **BSS Identifier**, uma vez que o AP fez o repasse do quadro , conterá o MAC do AP;
- **Endereço 3:** abstraído como **Sender Address**, uma vez que o *Host* A que originou o quadro, conterá o MAC do *Host* A.

A mesma analogia pode ser utilizada quando o *Host* B deseja enviar um quadro ao *Host* A.

2.7.2 Estação-*Gateway*

Este caso pode seguir a analogia do caso anterior, porém com algumas ressalvas. Primeiro, nota-se que agora há dois tipos de enlace no envio, culminando na coexistência do protocolo 802.11 e *Ethernet* no envio de dados.

Quando o *Host* A deseja enviar um datagrama ao *gateway*, irá encapsulá-lo em um quadro 802.11 idêntico ao descrito no processo de envio de quadro do *Host* A para B, porém trocando as referências do *Host* B pelo *gateway*.

Ao receber o quadro, o AP irá ler os cabeçalhos do quadro e, ao checar que o mesmo destina-se ao *gateway*, irá retirar o cabeçalho 802.11, substituindo-o por um cabeçalho *Ethernet*, com MAC de origem igual ao MAC do *Host* A e o MAC de destino igual ao do *Host* B. Por fim, envia o quadro ao *gateway*. Observe que neste processo, o AP fez o repasse, mas agiu como um meio transparente ao *gateway*, uma vez que seu endereço MAC não foi referênciado no cabeçalho *Ethernet*.

Mesmo assim, quando o *gateway* responde o *Host* A, o mesmo envia um quadro *Ethernet* ao AP, com MAC de destino igual ao MAC do *Host A* e MAC de origem igual ao MAC do *gateway*.

O AP por sua vez irá ler o quadro, desmontá-lo e remontá-lo na forma de um quadro 802.11, com o *bit From AP* e com os campos de endereços preenchidos da seguinte forma:

- **Endereço 1:** abstraído como **Destination Address**, uma vez que o *Host* A irá receber o quadro, conterá o MAC do *Host* A;
- **Endereço 2:** abstraído como **BSS Identifier**, uma vez que o AP fez o repasse do quadro , conterá o MAC do AP;
- **Endereço 3:** abstraído como **Sender Address**, uma vez que o *gateway* originou o quadro, conterá o MAC do *gateway*.

2.7.3 Recebimento do Quadro

Em ambos os casos mostrados, os destinos finais receberam quadros do tipo 802.11 que diferem em tamanho e conteúdo dos quadros *Ethernet*. Logo, aplicações deveriam ter de decifrar qual foi o tipo de quadro recebido para poder lê-lo corretamente.

Entretanto, durante o desenvolvimento do trabalho, foi constatado que todo pacote que era entregue ao Webspy tinha a assinatura *Ethernet*, mesmo que obtido por uma interface de rede sem fio. Depois de certa pesquisa [4], foi constatado que o *hardware/drivers*

de rede acabavam por realizar a tradução dos quadros 802.11 para quadros *Ethernet*. Consequentemente, a implementação sobre enlace sem fio acaba por ser transparente às aplicações, não sendo necessário o manejo de pacotes com a assinatura do 802.11 por parte do Webspy.

Uma vez que foi assumido que o *host* atacante já estará ingressado na BSS do *host* vítima, não é necessário se preocupar com protocolos de autenticação e cifragem de dados estipulados pela BSS, uma vez que o atacante já é uma estação autenticada e os *drivers* de rede tomaram conta da cifragem. Logo, estas responsabilidades também são transparentes à aplicação. Concluindo, a aplicação poderá ter o mesmo comportamento em enlaces com ou sem fio.

Capítulo 3

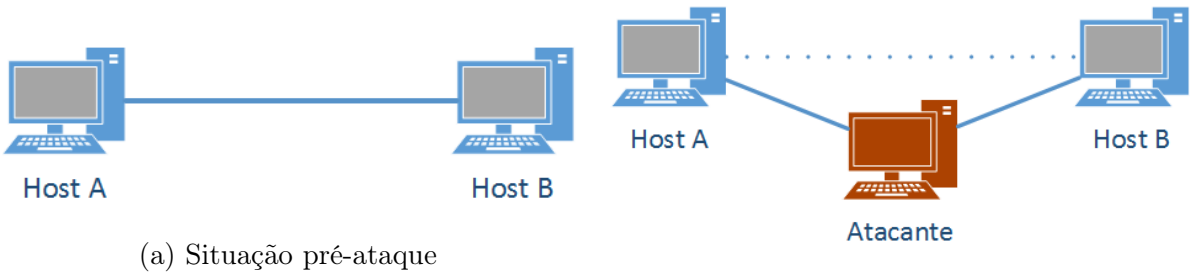
O *Man-in-the-Middle* e ARP *Spoofing*

O ataque *man-in-the-middle* é bem conhecido e explorado na literatura [16] [39]. É utilizado para um determinado atacante se infiltrar na comunicação entre duas vítimas, interceptando todo o tráfego, enquanto a comunicação entre ambas ainda é mantida pelo atacante. Consequentemente, o *man-in-the-middle* acaba sendo o primeiro degrau para uma série de ataques em rede.

Neste capítulo será dada a base desta técnica, com foco no seu uso baseado no protocolo ARP, o qual é nomeado de **ARP *Spoofing***. Este foi a base para o desvio do tráfego da ferramenta.

3.1 Princípio Geral

A base do ataque é a intrusão de um atacante em uma conexão entre duas vítimas, de modo que consiga interceptar todo o tráfego, podendo alterá-lo, omiti-lo ou simplesmente registrá-lo e, finalmente, repassa-lo aos destinos corretos, de forma a criar uma conexão virtual entre as vítimas, caracterizando um ataque silencioso. Os esquemas mostrados em 3.1 ilustram a lógica.



(a) Situação pré-ataque

(b) Situação pós-ataque

Figura 3.1: Esquematização do Ataque *Man-in-the-middle*

Quando a conexão entre vítimas é previamente cifrada, o atacante deverá tomar outras decisões para que consiga ter acesso aos dados interceptados, sendo obtendo a chave que decifraria a codificação dos dados ou então evitando que os dados sejam previamente cifrados. De qualquer maneira, neste ataque, as vítimas não devem percer invasor no meio da comunicação, uma vez que este deverá continuar o trânsito de informações, simulando uma conexão legítima.

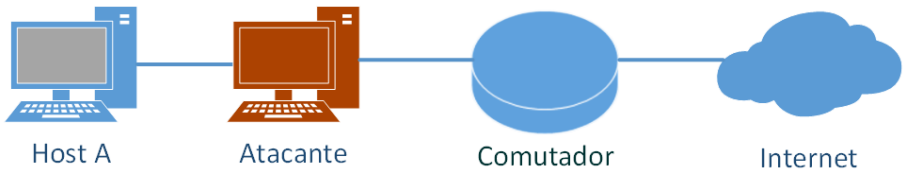


Figura 3.2: *Man-in-the-middle* para obtenção de tráfego *web*

Por esta perspectiva, o ataque pode ser realizado entre dois *hosts* quaisquer. Assim, se o tráfego *web* de uma máquina é desejado (sendo este o foco do trabalho), seria necessária a interceptação da comunicação entre um dado *host* e o comutador que traz os dados da *Internet*, ou seja, o *gateway*. Este esquema é mostrado na Figura 3.2.

As Figuras 3.3 e 3.4 exemplificam as situações pré e pós-ataque, nas quais o ataque em discussão neste trabalho deseja chegar. Inicialmente, a vítima requisita e recebe dados normalmente do *gateway*. Na segunda situação, o *host* atacante recebe os pacotes de dados endereçados logicamente à vítima, porém, fisicamente a ele. Em seguida, repassa os pacotes à vítima, de forma que ela não perceba que foram primeiramente enviados ao atacante.

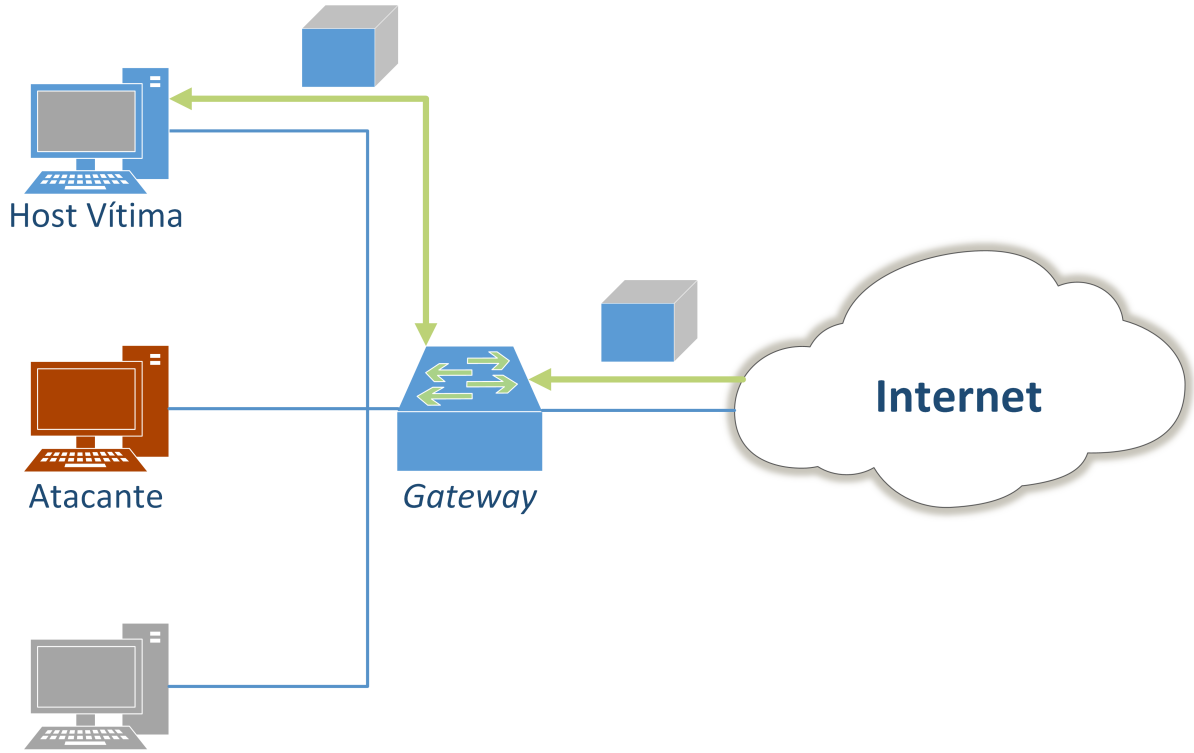


Figura 3.3: Sub-rede antes do ataque, com tráfego normal de dados

É possível perceber que nesta configuração um fato importante está ocorrendo: seguindo o protocolo explorado, o *host* vítima acredita que o *gateway* é o atacante e vice-versa. Esta é a premissa que permite ao atacante obter acesso ao fluxo de dados de ambas as vítimas, de modo imperceptível.

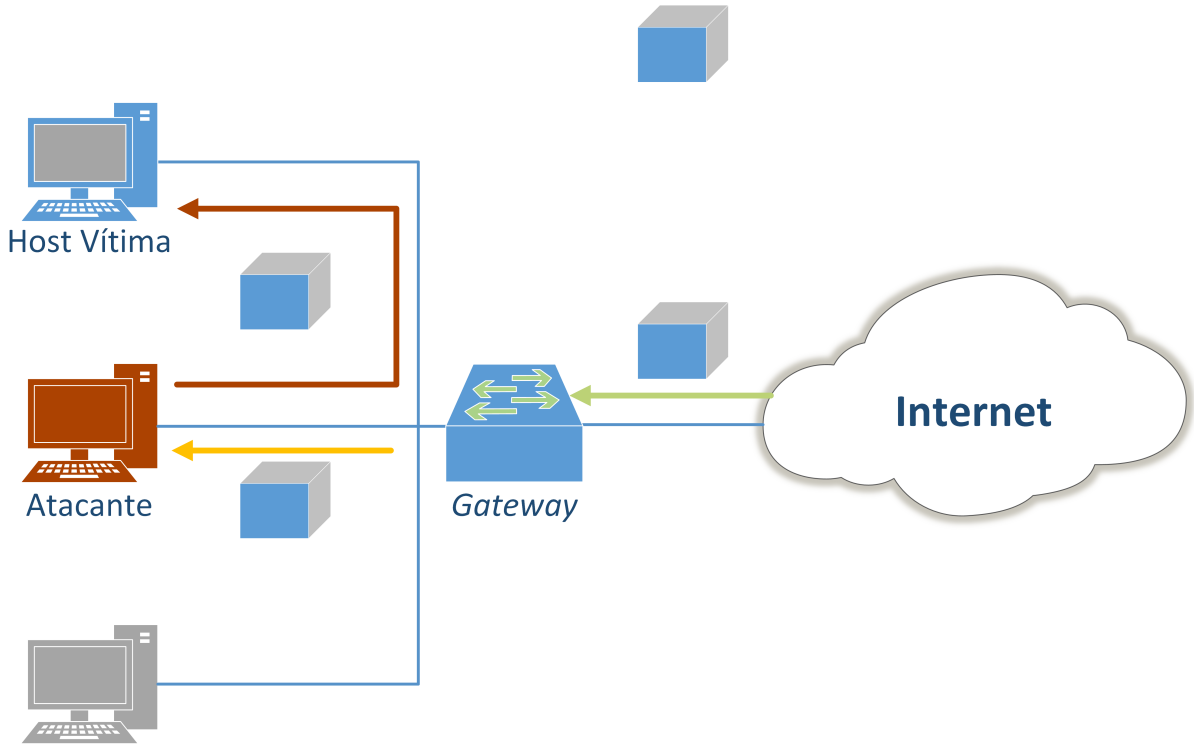


Figura 3.4: Sub-rede depois do ataque, com o tráfego de dados sendo interceptado

A seguir, veremos no ambiente de uma sub-rede, qual é a forma de se alcançar isto e como fazer com que um nó A desvie um pacote, que teoricamente deveria ir ao nó B, até a um nó C.

3.2 ARP Spoofing

Como abordado no Capítulo 2, para uma máquina enviar dados à outra em sua sub-rede, ela necessita de dois endereços do destino: o IP e o MAC. Mais especificamente, sempre que um nó for enviar dados a outro, ele utilizara o IP para realizar esta comunicação, criando os datagramas correspondentes. Ao serem enviados ao enlace, os datagramas serão envelopados pelo cabeçalho do protocolo de enlace (geralmente *Ethernet* ou 802.11), onde será inserido o MAC de destino, o qual, efetivamente, localiza o *host* na sub-rede. Neste processo, o remetente checa em sua tabela ARP a entrada correspondente ao IP de destino, para obter o MAC correspondente e preencher, corretamente, os cabeçalhos de enlace.

Entretanto, suponha que, na hora do envio do quadro, um *host* A consultasse sua tabela ARP procurando o MAC de um *host* B, utilizando IP correto de B, e encontrasse o MAC do *host* atacante, o *host* C. Desta forma, o quadro seria preenchido utilizando o MAC de C, porém, com o IP de B. Ao ser enviado pela interface de rede, o quadro acabaria chegando em C. A Figura 3.5 mostra a situação descrita.

Perceba que, neste processo, não houve nenhum comportamento atípico do envio de dados pela parte do remetente: os cabeçalhos foram corretamente preenchidos conforme as informações que o nó continha, mesmo que uma delas apresentasse uma informação

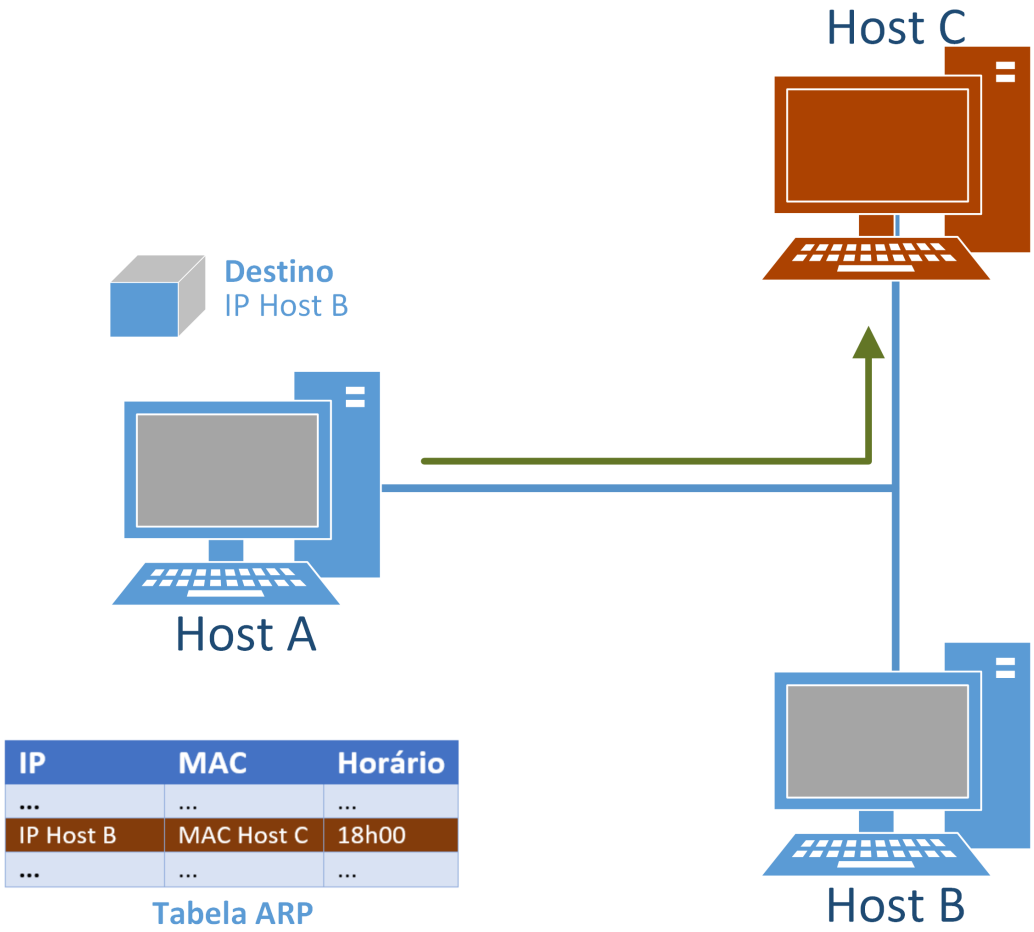


Figura 3.5: Sessão de *Spoofing* causada por uma entrada incorreta na Tabela ARP do *Host* A

incorreta em relação à topologia real da rede. Note também que, caso o mesmo ocorresse em B, todas as informações da conversa entre A e B seriam repassadas primeiro à C. Todo este resultado ocorreria em consequência de uma entrada incorreta nas tabelas ARP de A e B.

Esta é a base para o **ARP *Spoofing***, técnica de *man-in-the-middle* para o protocolo ARP, definido em [19], [41], [21], [28] e [23]. O ataque propõe que sejam criadas entradas falsas nas tabelas ARP dos *hosts* vítimas, por meio de pacotes ARP falsos, onde os IPs de ambos sejam mapeados para o MAC de um dado atacante, permitindo que este receba o quadros que as vítimas estão trocando. Concorrentemente, o atacante deve fazer o correto repasse entre os *hosts*, para que a comunicação entre eles flua.

Para chegar neste resultado, o ataque é bem definido em duas fases distintas: **ARP *Poison***, onde são criados os quadros ARP com as falsas associações entre IP de uma vítima e MAC do atacante, e enviados às respectivas vítimas e o ***relay***, fase onde o atacante deve repassar os quadros interceptados aos seus corretos destinos, para a manter a comunicação fluente entre as vítimas.

3.2.1 ARP *Poison*

Esta fase se apoia em dois fatos, já abordados no Capítulo 2, sobre o protocolo ARP:

1. **O protocolo não guarda estado**. A tabela ARP não guarda o estado de suas entradas, ou seja, não importa se a entrada apresentar informações discrepantes de iterações anteriores, a entrada será atualizada mesmo assim;
2. **A tabela ARP tem comportamento passivo**. A qualquer quadro ARP que apresente uma informação passível de atualizar a tabela, esta o fará.

Logo, aproveitando o fato 1, o atacante pode criar quadros ARP falsos, que contém uma associação entre IP e MAC incorreta, onde um IP de uma vítima A será associado ao MAC do atacante. Quando a vítima B (ou qualquer outro *host*) receber este quadro, irá atualizar de modo legítimo sua tabela.

O tipo de quadro ARP a ser enviado, para evitar o máximo de tráfego suspeito na sub-rede alvo, deve ser uma ARP *reply* pois, segundo o fato 2, o nó irá receber este quadro sem problema e prontamente aproveitá-lo em sua *cache*.

Tomando como exemplo o esquema da sub-rede base deste trabalho, na Figura 2.9, traça-se a forma de ataque sobre a vítima. Relembrando o cabeçalho do quadro ARP, em especial o ARP *reply*, este deverá conter todos os endereços MAC e IP preenchidos, sendo o primeiro par correspondente a quem responde e o segundo par a quem perguntou. Logo, ao forjar a resposta ARP falsa, o primeiro par deverá conter o IP da vítima A e o MAC do atacante e o segundo par deverá conter os endereços IP e MAC da vítima B, sendo esta a vítima a qual deseja-se criar a entrada falsa na tabela ARP. Para falsificar a entrada na respectiva tabela do *host* A, é executado um processo análogo.

Note que, o atacante deve ter um conhecimento prévio dos IPs das suas vítimas, para conseguir direcionar corretamente os quadros de *spoofing*. Além disso, é necessário ter cuidado com o cabeçalho de enlace a ser inserido. Neste, o enredeço MAC do remetente deve ser sempre o MAC do atacante, enquanto o endereço de destino deve ser o endereço da vítima à qual deseja-se corromper a tabela ARP. Transpondo a solução anterior para o exemplo usado neste trabalho, duas ARP *reply* deverão ser criadas:

- **Para o *host* vítima:** na resposta ARP, o par de quem responde deverá conter o endereço IP do *gateway* e o MAC do atacante. No par de quem recebe a reposta, deverão estar os endereços IP e MAC do *host* vítima. No cabeçalho de enlace, o MAC remetente é o do atacante e de destino o do *host*;
- **Para o *gateway*:** na resposta ARP, o par de quem responde deverá conter o endereço IP do *host* vítima e o MAC do atacante. No par de quem recebe a reposta, deverão estar os endereços IP e MAC do *gateway*. No cabeçalho de enlace, o MAC remetente é o do atacante e de destino o do *gateway*;

Ao fim deste processo, sempre que a vítima desejar enviar algo para o *gateway*, enviará para o atacante, de forma legítima e imperceptível. Analogamente, o mesmo acontece para o *gateway*. Dessa forma, o atacante sempre receberá os dados da comunicação entres os dois, agindo como um agente no meio da conversa. O esquema da Figura 3.6 demonstra as respostas forjadas sendo enviadas.

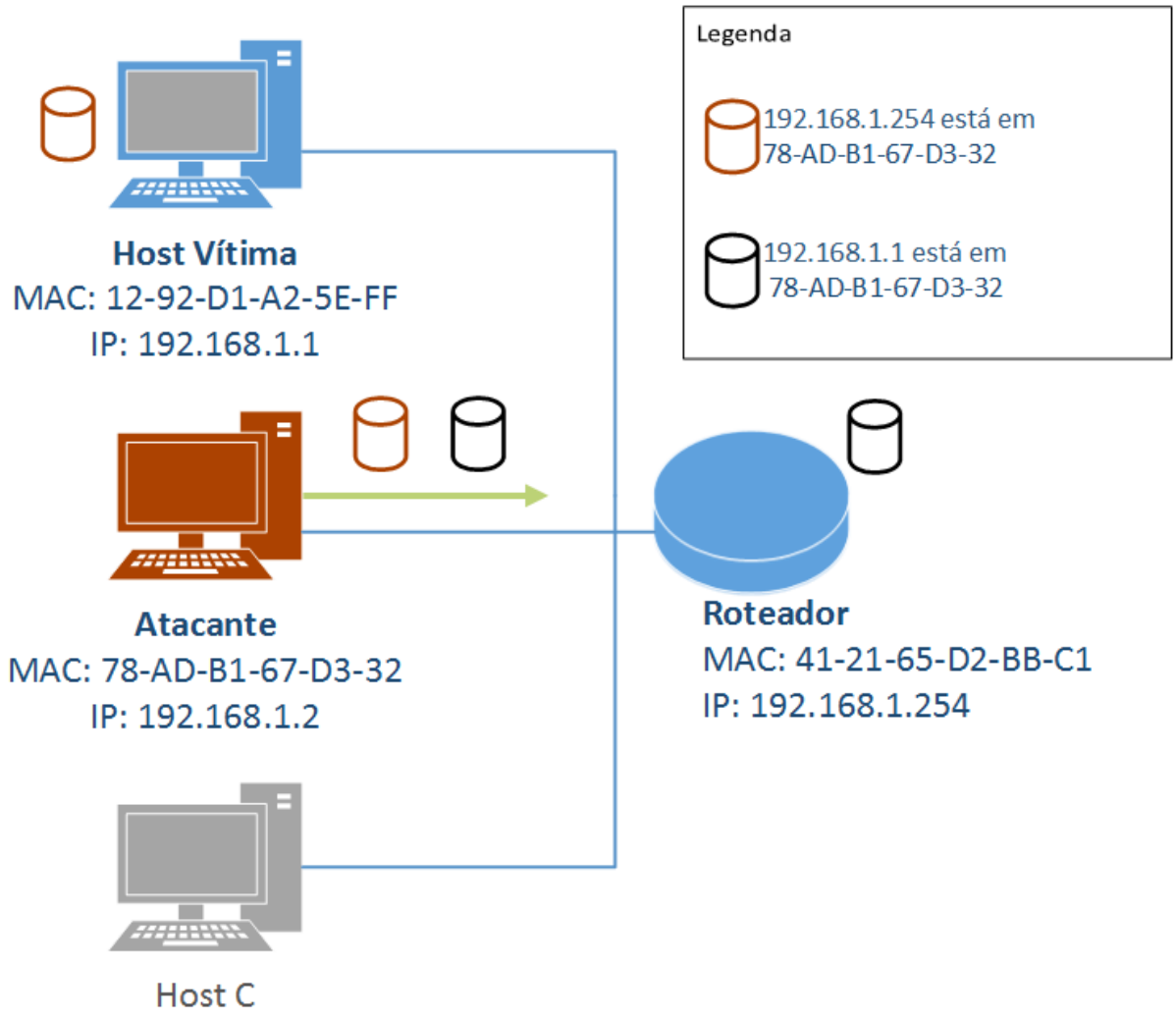


Figura 3.6: Envio dos dados forjados para as vítimas. Este processo deve ocorrer enquanto o ataque durar

3.2.2 *Relay*

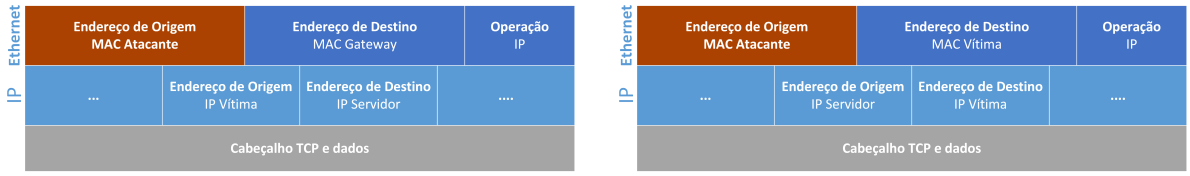
Finalmente, para completar o ataque é necessário que haja a fase de repasse dos pacotes, para que a comunicação entre as vítimas não seja quebrada. Esta fase é sutil, mas muito importante para o sucesso e a discrição do ataque pois, caso a comunicação entre os *hosts* seja cessada ou comprometida de alguma forma, pode levantar suspeitas.

A realização desta etapa pode ser feita tanto de forma manual como automática, ambas já citadas em [18]. A forma automática é baseada nas ferramentas embutidas nos sistemas operacionais para roteamento de pacotes segundo os protocolos TCP/IP, função que vem desabilitada por *default* nesses sistemas. Usando essas funções, o atacante tem menos poder sobre os pacotes, uma vez que todo o processamento dos mesmos é feito pela política TCP/IP do *kernel* do sistema.

Na forma manual, após a fase de *poison*, quando o atacante já recebe os pacotes das vítimas, um problema de endereçamento de enlace ocorre. Em um repasse de uma vítima A para uma B, o atacante não pode simplesmente repassar o pacote sem alterações, uma

vez que neste o remetente é o *host* A e o destino é o atacante. Caso o atacante enviasse este pacote ao enlace, o mesmo seria descartado pela vítima B, pois o MAC de B não está assinalado como destino no pacote.

Desta maneira, é necessária uma adaptação do quadro, sendo feita no próprio atacante, o qual deve zelar pela transparência do ataque. Seguindo a lógica do *spoofing*, para um pacote vindo da vítima A com destino à B, o cabeçalho *Ethernet* do quadro deve sair do atacante contendo o MAC de remetente preenchido com o MAC do atacante e o MAC de destino preenchido com o MAC da vítima B. O método é a mesmo de um pacote que sai da vítima B com destino à vítima A. Assim, o atacante passa despercebido no enlace, agindo como um meio transparente. As Figuras 3.7a e 3.7b mostram como é quadros repassados devem ser preenchidos pelo atacante.



(a) Pacote enviado do *gateway* para vítima (b) Pacote enviado da vítima para *gateway*

Figura 3.7: Pacotes de dados alterados para realização do *man-in-the-middle*

3.2.3 Considerações

Existem diversas técnicas de defesa contra o ARP *spoofing*, como [21], [28] e [23]. Mesmo assim, essas técnicas raramente são aplicadas em redes domésticas e, até mesmo, coorporativas, deixando estas subredes vulneráveis ao ataque. Esta falha é agravada pelo fato do ARP *spoofing* explorar vulnerabilidades de um protocolo essecial para a comunicação entre os *hosts* de uma rede.

Sob esta perspectiva, é necessário realizar algumas considerações sobre o ARP *spoofing* e a ambientação do mesmo, [18], [41] e [21]:

- É importante notar que, mesmo que o protocolo ARP não guarde estado, o mesmo guarda o tempo em que recebeu as respostas ARP, dando um prazo de validade para cada entrada na tabela ARP. Desta forma, é importante sempre estar realimentando as entradas falsas criadas nas vítimas, mandando periodicamente os pacotes forjados e garantindo que os endereços reais das vítimas não sejam usados em seus cabeçalhos de camada de enlace;
- Caso a sub-rede tenha algum protocolo de cifragem de dados, muito comum em redes *Wi-Fi*, o atacante deve seguir esta cifra, caso contrário, não conseguirá realizar a comunicação com os *hosts* vítimas desejados. Para tal, duas situações podem ser adotadas como saída: o atacante deve também ter acesso a alguma chave compartilhada prévia da comunicação ou ser um nó já autenticado na sub-rede. Neste trabalho, foi abordada a segunda situação;
- O *relay* pode ser um gargalo neste e em outros ataques que dependam do mesmo. Como o atacante é um novo agente na comunicação entre as vítimas, existe um novo

overhead no repasse do tráfego. Isso pode causar oscilações e atrasos na resposta, levantando suspeitas do ataque. Portanto, nesta fase é importante estar atento à perfomance.

Finalmente, agora com o dados da sessão entre as vítimas é possível prosseguir para a próxima fase do trabalho: a obtenção de tráfego *web* para visualização.

Capítulo 4

Relay: Lidando com HTTP e HTTPS

Após a conclusão do *man-in-the-middle*, o atacante receberá todo o tráfego entre o *host* vítima e o *gateway*. Neste processo, os pacotes de dados deverão ser filtrados de forma a obter apenas o tráfego *web*, de modo que o atacante passará a fazer a ponte da comunicação entre o cliente e servidor das páginas HTML.

Esse processo sendo feito de forma desprotegida não apresenta grandes desafios. Entretanto, a presença de protocolos de cifragem de dados na troca de informações na *Internet*, através do HTTPS por exemplo, configurou um problema a ser trabalhado. A cifragem das sessões *web* entre cliente e servidor causa um impasse para o atacante, uma vez que agora o mesmo tem de conseguir ler os dados cifrados para poder renderizá-los. O SSL, e mais adiante o TLS, impuseram novas barreiras ao ataque.

Neste capítulo será abordado o tipo de dado alvo da aplicação *Webspy*, e como a mesma irá contornar a cifragem de dados por conexões HTTPS. Para isto, será necessário um pequeno embasamento teórico em protocolos essenciais para conexões entre clientes e servidores *web*.

4.1 *Transmission Control Protocol*: o TCP

O TCP (*Transmission Control Protocol*) é um protocolo da camada de transporte, sendo definido em [26], [8] e [16], e determinante para o transporte de dados em redes.

Por definição, o TCP é o protocolo que cria canais confiáveis para a transmissão de dados em redes não confiáveis. Ele é um protocolo voltado à conexão, sendo ponto-a-ponto. Ou seja, há apenas uma interação entre dois *hosts* que devem antes estabeler uma conexão para iniciar a transmissão de dados. Ainda, o protocolo provê confiabilidade sobre esta, garantindo integridade, ordenação e completude dos pacotes transmitidos, por meio de temporizadores nos dois nós de transmissão, *buffers* e números de sequência em cada pacote enviado.

Quando dois processos em *hosts* distintos desejam se comunicar por um canal TCP, uma conexão será aberta através de seus *sockets*, estrutura responsável por criar canais de comunicação entre processos. Cada *socket* utiliza uma porta de comunicação, identificada por um número inteiro de 0 até 65535. No caso do TCP, as portas de comunicação dos *sockets* não precisam ser iguais. A partir destes elementos, dois processos podem abrir uma conexão para transferência de dados.

Neste trabalho não é essencial detalhar como o protocolo TCP garante a confiabilidade. O foco aqui é saber trabalhar com o mesmo para criar estas conexões, principalmente no que diz respeito à conexões que irão realizar o transporte de dados HTTP e HTTPS. Dessa forma, serão estudados a estrutura do cabeçalho TCP e o esquema de conexões entre dois sistemas finais, entendendo como manipulá-las.

4.1.1 Estrutura e Cabeçalho

Um dado pacote de dados enviado sobre TCP é denominado **segmento TCP**. Um segmento respeita o limite de tamanho de outros protocolos, sendo a principal a MTU. Desse modo, dados transmitidos que ultrapassem essas barreiras são quebrados em vários segmentos no remetente, sendo remontados no destino.

O cabeçalho de um segmento possui 20 *bytes* (em sua grande maioria), como mostrado na Figura 4.1. Este é dividido nos seguintes campos:

- **Porta de origem:** número da porta de comunicação no remetente;
- **Porta de destino:** número da porta de comunicação no destinatário;
- **Número de sequência:** número de identificação do segmento, utilizado para remontagem correta dos pacotes de dados extensos que são fragmentados em partes;
- **ACK:** verificador utilizado em conjunto com o número de sequência para garantir que nenhum segmento foi perdido nas transmissões;
- **Comprimento do cabeçalho:** indica o comprimento efetivo do cabeçalho TCP;
- ***Flags* de auxilio:** *bits* utilizados para o controle do protocolo. Estas indicam estados de execução de conexões e ações sobre as mesmas. A Tabela 4.1 mostra essas operações e ao que elas dão margem;
- **Tamanho da janela de recepção:** indica o tamanho do *buffer* de dados no *host* receptor para montagem dos pacotes de dados;
- **Ponteiro de urgência:** indica a posição, no segmento, de dados urgentes que podem ter sido anexados.

Flag	Código	Descrição
`SYN`	`0x02`	Requisita estabelecimento de um canal com a outra ponta
`ACK`	`0x10`	Confirma o recebimento de alguma operação ou dado
`PSH`	`0x08`	Sinaliza o envio imediato do conteúdo do *buffer*, mesmo que não esteja cheio
`URG`	`0x20`	Indica presença de dados urgentes no pacote
`FIN`	`0x01`	Finaliza uma conexão
`RST`	`0x04`	Reseta a conexão

Tabela 4.1: *Flags* do TCP

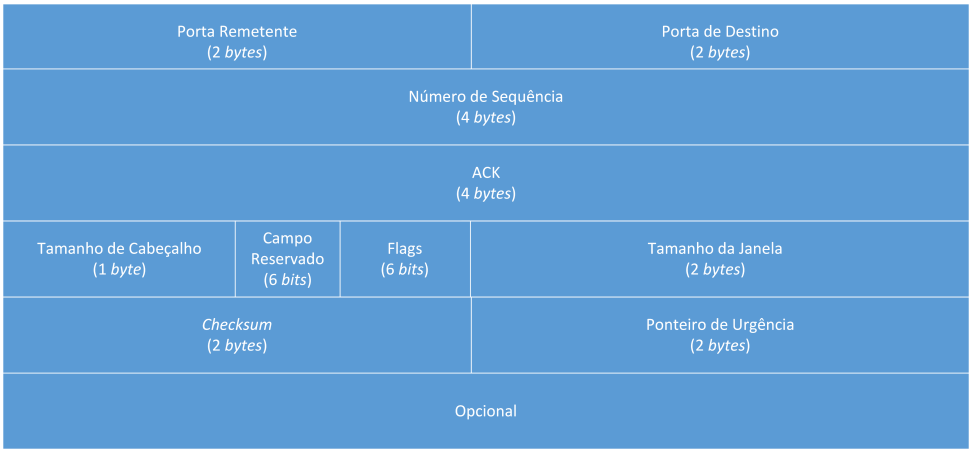


Figura 4.1: Cabeçalho TCP

4.1.2 Enviando Dados

Analisando o cabeçalho, percebe-se que cada segmento espera por uma porta origem e uma porta de destino. Esta é a forma de se multiplexar e demultiplexar dados em *hosts*, uma vez que o mesmo pode estar enviando e recebendo dados de diversas aplicações diferentes simultaneamente. Lembrando que a localização dos *hosts* em si é implementada pelos protocolos das camadas de rede e de enlace.

Segundo [8], dados HTTP, normalmente utilizados nos navegadores e servidores *web* atuais, possuem uma porta convencional para conexão, a porta 80, habilitada em servidores com o estado `LISTENING`, indicando que a mesma está aceitando conexões. Dessa maneira, navegadores iniciam a conexão abrindo uma porta qualquer, que estabele uma conexão com a porta 80 de determinado servidor *web*. Este processo é chamado de *bind*.

Estabelecida a conexão, o navegador pode realizar requisições ao servidor, sendo ele o cliente da sessão. Alguns dados requisitados, ou até mesmo enviados, podem ser grandes demais, logo serão fragmentados pelo TCP e enviados em segmentos distintos, para serem remontados no destino. Este processo é efetuado pelas operações do TCP, indicadas no campo de *flags* dos cabeçalhos dos segmentos. Ao fim da troca de dados, o cliente (navegador) solicita o fim da conexão e ambas as pontas finalizam a mesma.

Todos esses processos são parte de um complexo esquema do TCP para inicar, manter e finalizar conexões, controlado pelas *flags* de controle, números de sequência, ACKs e indicadores de tamanho de janela, os quais são demonstrado pelo diagrama de estados da Figura 4.2.

4.2 HTTP

Finalmente entendido como os dados são transferidos, será abordado o tipo de dado que desejamos manipular para renderizar as páginas da *web*: mensagens *Hypertext Transfer Protocol* - o HTTP - bem definido em [31].

O mesmo é um protocolo da camada de aplicação, utilizado na *web*, em navegadores, servidores e aplicações diversas. Por definição, é um protocolo destinado ao envio de

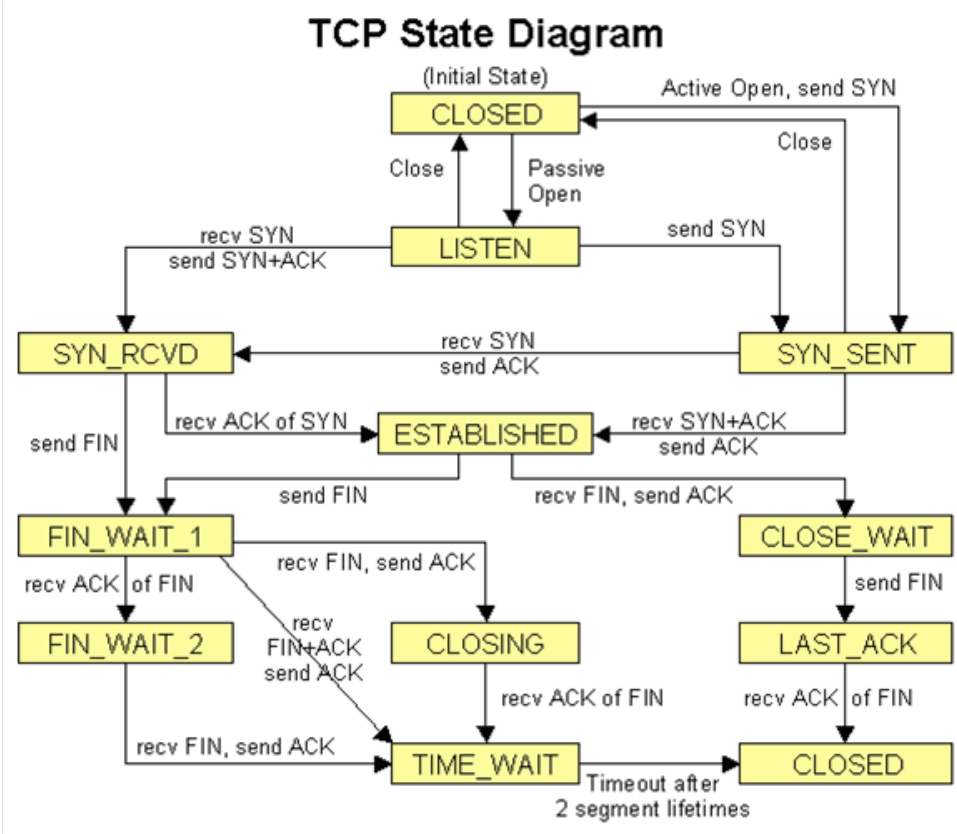


Figura 4.2: Diagrama de estado do protocolo TCP

dados de mídias diversas, apresentando uma tipificação e negociação do conteúdo enviado, permitindo a conversa entre aplicações com linguagens distintas.

O protocolo trabalha com a arquitetura cliente-servidor, ilustrada na Figura 4.3, trabalhando com requisições de objetos e respostas para estas. Neste trabalho, o foco será o conteúdo *web*, utilizado para montagem de páginas *web*. Dentro deste estão objetos como páginas HTML, imagens, arquivos CSS ou Javascript, entre outros.

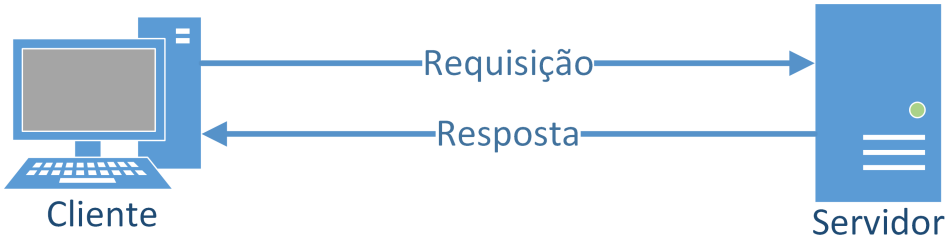


Figura 4.3: Topologia cliente-servidor do HTTP

Cada objeto é referênciado por um caminho único, chamado URL (*Uniform Resource Locator*). Este tem duas partes: uma relativa ao servidor hospedeiro do objeto, referenciado pelo endereço de rede, e a outra relativa à localização do mesmo na árvore de diretórios do servidor. Logo, as requisições feitas por um cliente são simplesmente requisições de arquivos. Desta forma, o HTTP traz o seu conteúdo, utilizando canais TCP para transmissão desses arquivos.

É importante lembrar que uma página *web* é composta por vários arquivos (os quais serão detalhados no Capítulos 5) e, portanto, várias conexões TCP são abertas, uma para cada arquivo. Essa estratégia é chamada de conexão não-persistente, presente em versões anteriores do HTTP. Entretanto, a mesma foi abandonada na versão 1.1 do HTTP, a mais utilizada atualmente, pelo excesso de *overheads* a cada objeto transmitido, onde deveria ser iniciado e finalizado um canal TCP para cada objeto. Dessa forma, trabalha-se atualmente com conexões persistentes, onde todos os objetos de uma página são transmitidos em uma única sessão TCP, diminuindo este *overhead* no carregamento das páginas *web*.

4.2.1 Estrutura e Cabeçalho

O HTTP é composto por mensagens, que podem ser requisições ou respostas. Cada uma destas possuem uma apresentação diferente e campos específicos opcionais, o que dá uma característica muito dinâmica ao protocolo. Outro aspecto interessante, é que mensagens HTTP são escritas em codificação ASCII pura, sendo facilmente lidas por um usuário comum. Por isso, cada campo de uma mensagem é divido em chave e valor, separados por dois pontos (:). Os campos são apresentados em linhas diferentes, onde a mudança de linha é feita pelo *carriage return* (`\r\n`). O conteúdo da mensagem, ou seja, os dados que esta carrega efetivamente são separados por duas quebras de linha.

Uma mensagem de requisição HTTP tem no mínimo uma linha, que é a **linha básica da operação**, podendo ter diversas outras linhas que são denominadas **linhas de cabeçalho**, que trazem informações adicionais. Logo em seguida, vem o corpo da mensagem, o qual pode conter dados enviados pelo cliente da sessão, tais como formulários, *upload* de arquivos e outros. A linha inicial, ou linha de requisição, contém três campos importantes:

- **Operação:** contém a operação do HTTP. A lista de operações será descrita posteriormente;
- **URL:** arquivo sendo requisitado ao servidor, sendo apenas o caminho da árvore de diretorios no hospedeiro;
- **Versão do Protocolo:** campo autoexplicativo, onde é apenas ilustrado a versão do HTTP sendo utilizada.

Uma mensagem de resposta, por sua vez, traz uma estrutura semelhante, porém com alguns campos diferentes. A primeira linha, chamada linha de estado, traz os seguintes campos:

- **Versão do Protocolo:** análogo ao campo de versão da mensagem de requisição;
- **Código de Estado:** código de resposta à requisição do cliente;
- **Mensagem de Estado:** mensagem correspondente ao código anterior.

4.3 *Relay* Simples

Finalmente entendido o tipo de dado alvo pelo atacante e como este é direcionado entre cliente (vítima) e servidor, é possível traçar um esquema simples de *relay*, onde o atacante também poderá usar os dados para renderizar as telas da vítima.

O processo de *relay* para conexões HTTP simples pode ser observado na Figura 4.4 e pode ser dividido em 4 fases:

1. **Requisição do cliente:** inicialmente, a vítima irá fazer a requisição por uma página *web*. É importante notar que este tipo de requisição abre margem para diversas outras, uma vez que uma página não é só composta por texto, mas por imagens, arquivos auxiliares (CSS e Javascript) e outros arquivos de mídia. Estes porém vêm embutidos dentro da requisição da página. Esta requisição será endereçada ao servidor, porém irá primeiro passar pelo atacante;
2. **Repasse da requisição:** ao receber a requição, o atacante fará suas adaptações para passar despecebido pelo ataque. Em seguida, a requisição é enviada ao *gateway*, que faz seu intermédio enviando a mesma para a rede, onde o servidor de destino será alcançado.
3. **Resposta do servidor:** o servidor processa a requisição do cliente e envia sua resposta. É importante notar que, para o servidor, tudo está normal, pois recebe o IP da vítima como remetente. Esta resposta irá eventualmente chegar ao *gateway* da sub-rede e será repassada ao atacante;
4. **Repasse da resposta:** no atacante, os dados recebidos do servidor têm dois destinos: o navegador do atacante e a vítima propriamente dita. Para o processo de *relay*, só interessa o segundo. Assim, os dados podem ser replicados, e uma das partes ser imediatamente repassada à vítima.

Figura 4.4: Processo de *relay* simples

Observando o processo, nota-se que o mesmo não traz grandes *overheads*: é um processo simples de repasse, onde o gargalo está situado no momento do repasse do atacante para a vítima, onde os dados recebidos do servidor devem ser utilizados também no atacante, para reprodução das telas. Neste quesito, o ataque é simples e é possível manter um bom desempenho.

Entretanto, será verificado que, atualmente, a troca de dados ao longo da Internet já não funciona dessa forma. Devido às numerosas tentativas de ataques ao longo dos anos e do aumento da quantidade de dados de grande valor trafegados pela rede, foi necessário o uso da criptografia para garantir proteção aos usuários.

4.4 Cifragem e HTTPS

Constantemente, usuários acabam por realizar operações na Internet onde são usados dados pessoais e informações valiosas, além de existirem situações onde é necessária a proteção das informações transmitidas. Estes casos trouxeram a tona à importância da segurança da informação nos processos de transmissões de dados e logo, surgiram protocolos seguros para tais transmissões.

A *Internet*, atualmente, utiliza um padrão em larga escala, que recebe grandes investimentos anuais. Este está cada vez mais presente, até em conexões que não necessitam propriamente de uma cifragem com proteção robusta. Este padrão é o SSL, que mais tarde foi adaptado e documentado pelo IETF no TLS.

4.4.1 SSL

O SSL, sigla para *Secure Socket Layer*, é um protocolo da quarta camada, de transporte, porém largamente implementado na camada de aplicação, é definido e explorado em [16] e [33]. Age como uma implementação de segurança, com checagem de integridade e autenticação dos agentes da comunicação. Possui um esquema de apresentação muito semelhante ao TCP, divido em três etapas distintas: apresentação, formação de chaves e transferência de dados.

Historicamente, o SSL foi definido para dar uma proteção a conexões HTTP, que funcionam por texto puro. Agora, o início de uma conexão com o servidor irá implicar em uma sessão SSL, onde esta será utilizada apenas uma vez para uma cada sessão HTTP. O protocolo tem forte base no uso de chaves públicas e privadas, presentes nas duas pontas da comunicação (servidor e navegador), conjutamente com chaves simétricas, obtidas por meios da combinação das chaves públicas e privadas, com cifras aleatórias geradas a cada sessão de transmissão de dados.

Atualmente, o SSL está na versão 3.1, mais robusta e com base em vários protocolos criptográficos. Além disso, apresenta um esquema de proteção a ataques de falsificação de mensagens. O estabalecimento de uma sessão SSL é feito em etapas e envolve várias técnicas, de modo que o protocolo em si é composto por 5 subprotocolos a serem destrinchados a seguir.

SSL *Handshake Protocol*

O protocolo definine o estabelecimento de uma sessão SSL, para início da comunicação segura entre duas entidades. Inicialmente, a sessão irá começar com uma conexão TCP simples, sendo esta o canal a ser protegido. Uma vez implementado, o estabelecimento da sessão SSL se dá de modo semelhante ao *3-way handshake* do TCP. A seguir, será apresentado um esquema básico de como o SSL funciona, o qual será ilustrado pela Figura 4.5:

1. Inicialmente, o cliente emite uma mensagem chamada *ClientHello*, mostrando que deseja iniciar a sessão com o servidor. Esta mensagem traz três campos:

   - **Versão do Protocolo:** indicando qual a versão o está propondo à conexão;
   - **Número aleatório (*nounce*):** gerado a partir de um *timestamp* de 32 *bits* e um número aleatório de 28 *bits*. Este é utilizado na hora da derivação das chaves da sessão, para evitar ataques que queiram ter acesso a esse processo;
   - **Protocolos suportados:** cifra que mostra quais os protocolos criptográficos que o cliente suporta.

   Em versões superiores, outros dois campos também estão presentes: uma cifra de algoritmos de compressão suportados e um ID de sessão, que irá indicar se o cliente quer retomar uma sessão SSL antiga ou iniciar uma nova;

2. Ao receber a requisição, o servidor irá responder ao cliente com as seguintes mensagens:

   - ***ServerHello*:** resposta à requisição do cliente, onde é enviado o algorítmo criptográfico escolhido, o *nounce* do servidor e o ID da sessão SSL;
   - ***Certificate*:** quando o servidor deseja se autenticar (quase sempre), o mesmo envia o seu Certificado de Autenticação, o qual contém sua chave pública e certificados correspondentes.
   - ***ServerHelloDone*:** indica que o servidor enviou todas as mensagens que deveria enviar.

   Caso deseje, o servidor pode ainda enviar a mensagem *CertificateRequest*, onde exige que o cliente também se autentique. Esta traz os tipos de certificados aceitos pelo mesmo e a lista de autenticadores autorizados.

3. Ao receber as mensagens, o cliente pode gerar o Segredo Pré-Mestre (PMS - *Pre-Master Secret*). Este é obtido a partir da chave pública enviada no certificado do servidor, que que garante sua autenticidade. Gerada a PMS, o cliente envia as seguintes mensagens ao servidor:

   - ***ClienteKeyExchange*:** envio da PMS, previamente cifrada com a chave pública do servidor recebida pelo cliente. O servidor poderá decifrar a PMS utilizando sua chave privada, resultando na Chave Mestra (MS - *Master Secret*), que será a chave compartilhada da operação de cifragem;
   - ***CertificateVerify*:** ao validar o certificado do servidor, o cliente deve confirmar esta operação, enviando o *CertificateVerify* ao servidor;
   - ***ChangeCipherSpec*:** indica o alinhamento com o algorítmo de cifragem escolhido pelo servidor;
   - ***Finished*:** indica o fim da troca de mensagens entre cliente e servidor, definindo o sucesso da sessão. Esta é enviada de forma cifrada, segundo o algorítmo escolhido.

Caso o servidor tenha demandado autenticação do cliente, este ainda envia a mensagem do tipo *Certificate*, semelhante à enviada pelo servidor, contendo seu certificado público e *nounce*.

4. Por fim, o servidor envia duas últimas mensagens: *ChangeCipherSpec*, que é o alinhamento ao protocolo criptográfico escolhido, e *Finished*, que é a indicação do sucesso no estabelecimento do canal, sendo enviada de acordo com a cifra escolhida.

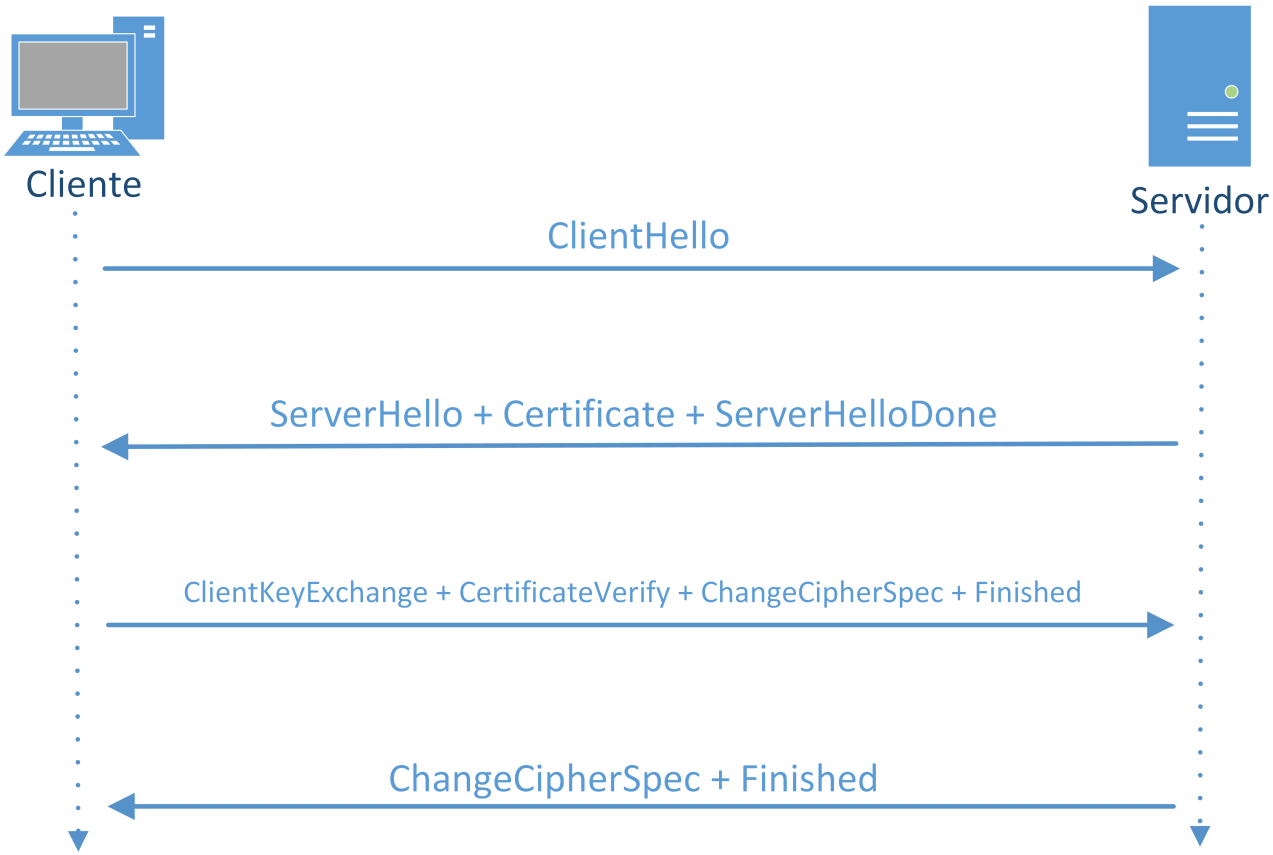


Figura 4.5: Processo de estabelecimento de sessão SSL

Por fim, cliente e servidor derivaram a MS, sendo que o cliente usa o certificado público do servidor e o servidor a sua chave privada. Dentro destes, os *nounces* também podem ser utilizados. A Figura 4.6 mostra como é feita esta derivação da MS final.

A MS pode também ser utilizada para criar outros tipos de chaves, dependendo do algorítmo criptográfico utilizado. Os algoritmos criptográficos suportados pelo SSL são mostrados em [33]. Além disso, a PMS é utilizada para a implementação do MAC (*Message Authentication Code*), utilizado para garantir a autenticação dos nós.

SSL *Record Protocol*

Este protocolo lida com o encapsulamento dos dados a serem transmitidos pelo canal, ou seja, o preparo dos dados da aplicação para serem enviados através do canal seguro. O processo de montagem é ilustrado pela Figura 4.7 e o resultado final pela Figura 4.8

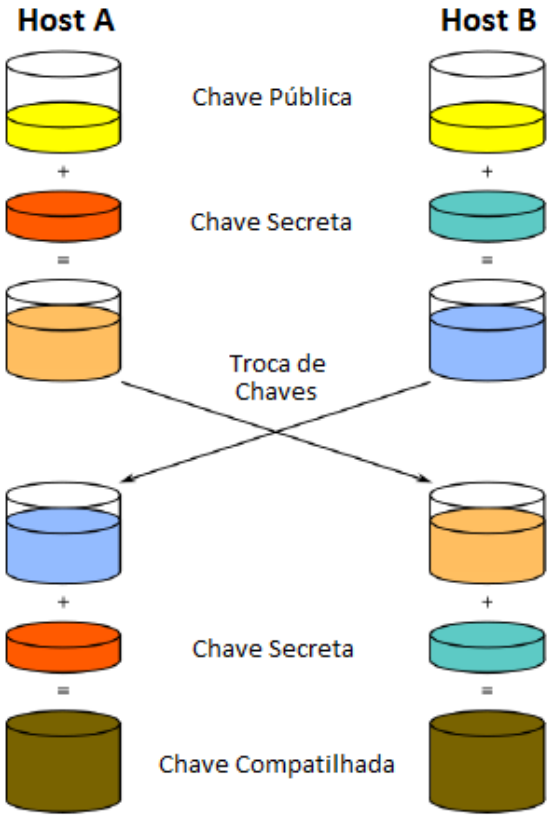


Figura 4.6: Processo de derivação da Chave Mestra (MS) entre duas pontas numa sessão SSL

Para isto, inicialmente, divide-se os dados em blocos chamados fragmentos, com tamanho de até 2^{14} *bytes*, envelopando-os em uma estrutura denominada *SSLPlainText*. Esta estrutura pode sofrer compressão, indicada no estabelecimento da sessão, passando o fragmento a se chamar *SSLCompressed*.

Aqui, o par de algorítimos para autenticação e cifragem já foi definido, sendo especificado na *cipher suite*. Logo, o código autenticador é calculado, por meio da função de *hash* definida na *cipher suite*, ao qual se dá o nome de MAC (*Message Authentication Code*).

Assim, os dados concatenados ao MAC são cifrados, segundo o algorítmo definido na cifra, e é obtido o fragmento *SSLCipherText*. Por fim, é adicionado um cabeçalho composto pelos seguintes campos:

- **Versão:** indica o protocolo de segundo nível do SSL ao qual se referem os dados da estrutura;
- **Tipo:** versão do protocolo SSL sendo utilizada;
- **Tamanho:** comprimento em bytes do fragmento *SSLCipherText*.

SSL *Change Cipher*

É responsável pela sinalização de alterações nas cifras criptográficas em uma comunicação em andamento, de modo que mensagens subsequentes tenham uma cifragem distinta

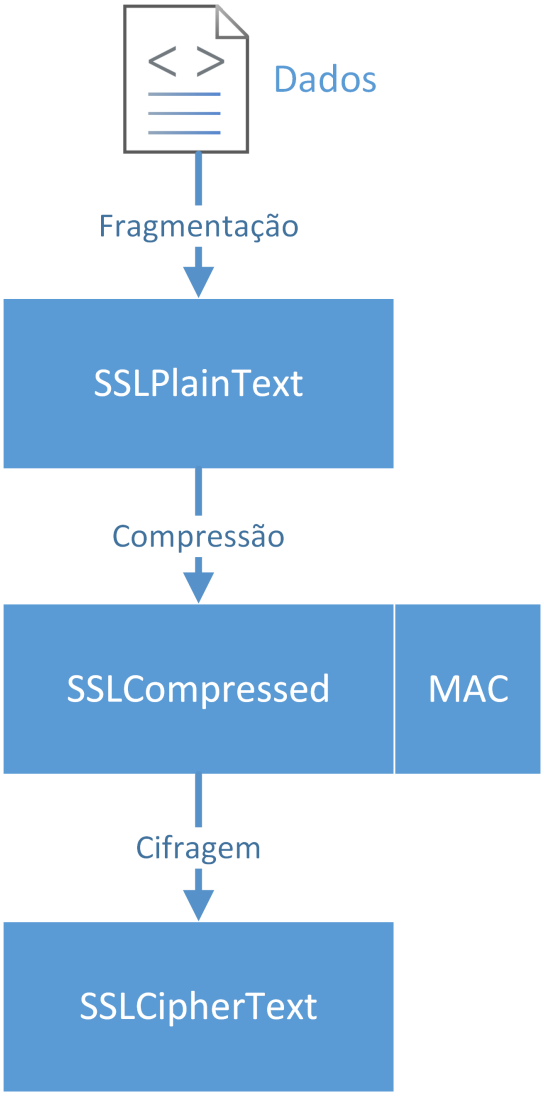


Figura 4.7: Processo de montagem do pacote SSL do *Record Protocol*

Figura 4.8: Pacote final do protocolo SSL a ser trafegado

das anteriores. A mensagem *ChangeCipherSpec* é o núcleo deste protocolo, indicando qual deve ser o novo algorítmo a ser utilizado.

SSL *Alert Protocol*

Sempre que alguma mensagem ou dado for verificado como incorreto ou estranho na comunicação, o SSL emite alertas, que invalidam a mensagem conflitante ou fazem a sessão tomar outro rumo. Este protocolo é responsável pela detecção e tratamento de erros dentro do SSL. Um alerta sempre é emitido pelo nó que detectou a falha, tendo um

nível de criticidade: pode ser um erro carregando um avisou ou um erro fatal, onde neste último a sessão é finalizada a fim de manter sua segurança.

SSL *Application Data Protocol*

A responsabilidade deste é a nível de camadas superiores, onde o protocolo lida com a coleta e envio dos dados provenientes das mensagens de aplicação para o SSL *Record Protocol* e vice-versa, de modo a garantir a integridade da interface entre as camadas de transporte e aplicação.

Dessa forma, na transmissão efetiva de dados , o protocolo SSL age da seguinte forma:

- **No remetente:** quando uma mensagem for enviada, a mesma pode ser comprimida, aplica-se um MAC sobre ela, cifra-a e, ao fim, a mesma é transmitida;
- **No destinatário:** quando é recebida pelo destino, a mesma é decifrada, verificada, sofre descompressão e, finalmente, é enviada às camadas superiores, às aplicações.

Quando uma conexão deseja ser finalizada, ou seja, quando um cliente quer desmanchar a sessão, o mesmo envia uma mensagem de finalização ao servidor, o qual atende e invalida qualquer outra mensagem relativa àquela conexão. Esta mensagem é necessária, ainda de forma cifrada, para evitar que atacantes finalizem a sessão entre os dois *hosts*.

4.4.2 TLS

O TLS (*Transport Layer Security*) surgiu como uma adaptação da última versão do SSL, definido em [38] na sua versão 1.0. Segundo estudos [37], o SSL 3.0 (base para o TLS 1.0) e o TLS 1.0 apresentam falhas críticas para a segurança de diversas aplicações, portanto, sua utilização deve ser evitada. Atualmente, o TLS já se encontra em versões superiores, que corrigem várias falhas da primeira versão e é adotado pela grande maioria dos navegadores e aplicações que desejam utilizar a cifragem com base em SSL, sendo recomendado por diversas entidades. Além disso, o SSL era um protocolo proprietário do navegador *Netscape*, sendo definido por esta entidade. Desta forma, o IETF definiu o TLS, a fim de normalizar e torná-lo um padrão a ser disseminado pela literatura.

Assim como o SSL, o TLS tem como objetivo garantir a segurança criptográfica em conexões entre duas partes, provendo uma forma transparente para as aplicações poderem se comunicar a partir de um padrão e criar um protocolo unificador, que evite a criação de novos protocolos que podem trazer novas falhas à segurança das comunicações. Por fim, também foca na garantia de privacidade, autenticação e integridade dos dados.

As diferenças entre O TLS e o SSL residem em detalhes técnicos, uma vez que o cabeçalho do TLS é muito semelhante ao do SSL. Algumas diferenças básicas residem nos protocolos criptográficos suportados entre ambos:

- o TLS possui uma extensão maior de protocolos, além de implementar o HMAC (*keyed-Hashing Message Authentication Code*) ao invés do MAC, onde no primeiro uma função de *hash* é aplicada a cada mensagem transmitida no protocolo;
- o TLS permite o uso de outras portas para comunicação entre cliente e servidor;

- Para iniciar o fluxo de dados, o servidor deve emitir uma mensagem de *Finished*, que deve chegar ao cliente. Ao contrário do TLS, no SSL este passo pode ser omitido;
- o TLS dá uma nova abordagem ao envio de Certificados Autorizados, onde não é mais necessário que a raíz de destino (servidor) tenha que oferecer o seu certificado, mas alguma instância acima deste, ou seja, nós anteriores ofereçam o certificado, permitindo o que é chamado de autoridade intermediária.

Por estas e outras razões mais específicas, os dois protocolos, TLS e SLL, sofrem de uma condição de não interoperabilidade, ou seja, não podem coexistir na mesma sessão.

4.4.3 Do HTTP ao HTTPS

As subseções acima mostraram como é possível garantir canais seguros em termos de cifragem de dados sobre um canal livre desta proteção. Quando clientes requisitam páginas da *web*, estes provavelmente estarão protegidos por conexões seguras, o que é um claro empecilho para o ataque *man-in-the-middle* pois, neste caso, o atacante não iria conseguir ler os dados interceptados, uma vez que estes estariam cifrados. A Figura 4.9 retrata essa situação.

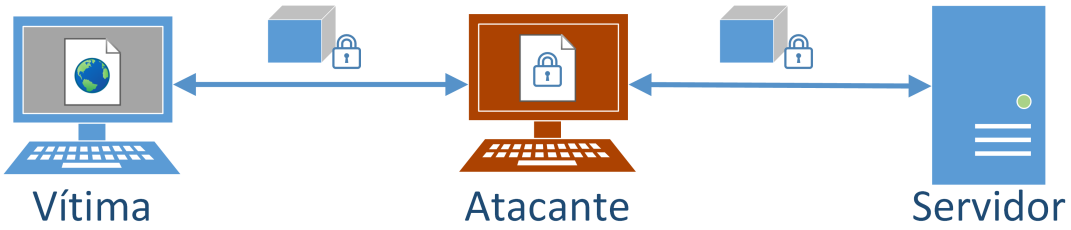


Figura 4.9: Esquema mostrando o atacante realizando o *relay* dos dados, porém não conseguindo visualizar as telas da vítima, por estas estarem cifradas

Nos navegadores modernos, o indício de que uma sessão *web* está sendo assegurada por uma sessão SSL está na barra de endereço da página requisitada, que passa de HTTP para o HTTPS, além de um sutil aviso para o usuário, como, por exemplo, um pequeno cadeado na barra de endereço. Entretanto, é interessante notar que usuários não tendem a requisitar páginas utilizando `https` no início dos endereços. Nem a inserção do `http` é algo esperado, uma vez que a ação padrão de usuários é apenas digitar o endereço do site que deseja acessar, por exemplo `unb.br`. Neste formato, uma sessão SSL não pode ser definida, pois não houve uma definição clara de que a mesma era necessária.

Assim, é preciso traçar por onde o HTTPS começa, entendendo como e quando uma requisição deixa de ser HTTP para ser HTTPS, solicitando o uso da segurança SSL. Segundo [34], existem 4 formas principais de ocorrer o redirecionamento da sessão HTTP para HTTPS:

1. **Resposta 302 do servidor:** clientes requisitam páginas HTTP normalmente. Entretanto, diversos servidores estabelecem uma regra de garantir que a mesma seja apenas respondida com informações via HTTPS. Para tal, o servidor emite a resposta HTTP 302 Found (Encontrado), porém implementada como uma resposta

303 See Other. Ou seja, o servidor envia um redirecionamento ao cliente, fazendo com que o mesmo busque a mesma página, porém por uma sessão HTTPS;

2. ***Hyperlinks* diretos:** dentro de páginas HTML, existem determinados objetos chamados âncoras, os quais são *hyperlinks* para outras páginas da *web*. Normalmente, quando um cliente recebe uma determinada página, seus *hyperlinks* já estão marcados com o HTTPS, para garantir que a requisição primária já seja segura;
3. **Requisição embutida em página *web***: quando uma página *web* é requisitada e enviada ao cliente, é possível que esta contenha outros objetos como arquivos CSS e Javascript, imagens e outros arquivos de mídia. Estes terão seus caminhos especificados na página e fica a cargo do navegador do cliente requisitar cada um deles. Normalmente, a URL para que o cliente possa requisitá-los já vem com a marcação do HTTPS, fazendo com que o cliente requisite primariamente uma sessão SSL;
4. **URLs diretas:** normalmente presente em links obtidos pela barra de favoritos de um navegador ou *bookmarks* ou simplesmente digitando na barra de endereços do navegador.

Outro aspecto importante a se notar é a porta de conexão usada para o HTTPS. Assim como definido, a porta padrão para conexões HTTPS é a 443, na qual o servidor estará escutando ativamente, aceitando requisições. Ao receber uma requisição, primeiramente estabelece a sessão TLS com o cliente e responde a requisição HTTP inicial.

Finalmente, chega-se à questão de como enfrentar o SSL na ferramenta Webspy. Como visto, é possível que o atacante tenha os dados cifrados disponíveis para ele. Porém, neste cenário, o mesmo teria que, previamente, obter a chave privada de um dos nós da comunicação, o que não é uma alternativa viável. Alternativamentem, o atacante poderia decifrar por meio de um ataque de força bruta cada mensagem interceptada, o que implicaria em um desgaste muito grande para a renderizar as telas. É necessário perceber também que as telas têm que ser renderizadas em tempo real.

O mais interessante, neste caso, seria evitar o SSL, ou seja, evitar que haja um sessão HTTPS robusta estabelecida entre vítima e servidor. Entretanto, ainda é necessário criar no servidor a ilusão de uma conexão HTTPS está sendo mantida, para que este possa validar a conexão com a vítima/atacante. Para tal, é apresentada a estratégia do SSL *Stripping*.

4.5 SSL *Stripping*

Apresentado em 2009, definido em [22], o SSL *Stripping* é uma técnica que não bate de frente com o SSL, mas o contorna, através do HTTP.

Inicialmente, a abordagem proposta era o SSL *Sniff*, técnica com abordagem *man-in-the-middle*, onde o atacante interceptava requisições HTTPS do cliente, gerava um certificado, obtido por um nó autenticado anterior ao servidor, assinando este certificado de forma inválida. Logo, este conseguia acesso à requisição do cliente e a repassava ao servidor. Entretanto, este repasse seria feito por meio de uma conexão HTTPS válida entre atacante e servidor, ou seja, o atacante agia como um *proxy*. Logo, para o servidor, a conexão estava totalmente legal. A Figura 4.10 ilustra a base do ataque.

Figura 4.10: Esquema básico do SSL *Stripping*

Um campo específico dos certificados é o `BasicConstraint`, o qual indica se o certificado em questão é assinado e validado por uma autoridade certificadora válida. O ataque, então, se baseava na omissão de várias aplicações em checar este campo ao receber um certificado qualquer. Por não checar este campo, um certificado com chave pública válida, mas assinatura de entidade de certificação inválida,acabava por ser aceito pelo cliente. Logo, o atacante poderia apenas resgatar a chave pública do domínio requisitado, criar uma assinatura teoricamente válida para o mesmo e obter acesso a requisição do cliente. O esquema da Figura 4.11 mostra como ocorre o ataque.

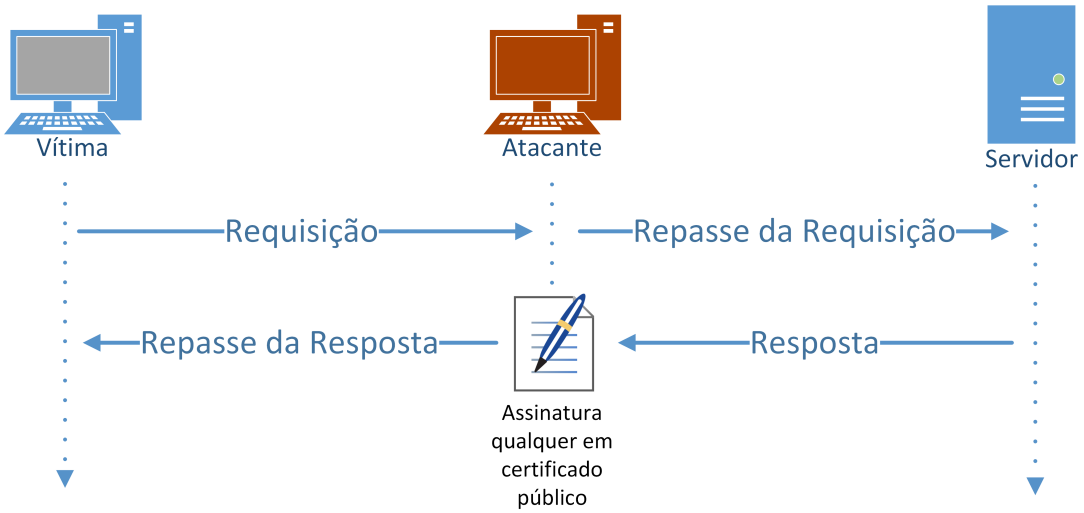


Figura 4.11: Esquema do ataque de SSL *Sniff*

Quando publicada esta falha, os detentores das aplicações logo a corrigiram, fazendo com que os navegadores passassem a checar o campo de `BasicConstraint`. Assim, quando um navegador detectava um certificado assinado por uma entidade não certificada, emitia um alerta ao usuário, indicando um certificado não autorizado. Então, o usuário estaria avisado do perigo que estaria correndo e poderia não prosseguir com a operação, comprometendo o ataque.

Dessa forma, surgiu o SSL *Stripping*, técnica onde o ataque se direciona primeiro à conexão HTTP. Como abordado na Seção 4.4.3, normalmente, por meio de uma conexão HTTP inicial é que se chega a uma conexão HTTPS. Portanto, o ataque propõe que toda a tentativa do cliente de atingir o estabelecimento de uma conexão HTTPS seja

interrompida. Para tal, basta agir sobre os pontos onde uma conexão passa de HTTP para HTTPS, abordados na sessão anterior.

Relembrando os métodos de como uma conexão passa de HTTP para HTTPS, dois métodos podem ser destacados: o redirecionamento por mensagem 302 ou por meio de uma requisição direta em HTTPS.

Para a primeira situação, tem-se os seguintes passos:

1. O cliente requisita uma determinada página *web*, por meio de uma requisição HTTP;
2. O atacante, já agindo como atravessador, recebe esta requisição e repassa ao servidor;
3. O servidor recebe a requisição e responde com o código 302, porém implementado como 303, demandando que a requisição seja refeita por meio de HTTPS;
4. O atacante recebe o redirecionamento. Inicialmente, ele estabelece uma conexão SSL/TLS com o servidor e requisita a página sobre este canal. Ou seja, a página é requisitada sobre HTTPS entre atacante e servidor;
5. O servidor agora responde à requisição, enviando a página requisitada, sobre o canal seguro;
6. O atacante recebe os dados e decifra os mesmos, recebendo, assim, a página completa (arquivos CSS, JS, imagens, etc.). Substitui todas as menções à URLs em HTTPS por HTTP e envia a página à vítima.

Na segunda abordagem, o cliente irá requisitar, inicialmente, a página por HTTPS. Esta mensagem porém, é uma mensagem HTTP, que contém a URL indicando a conexão HTTPS. Desta forma, quando o atacante receber a requisição, basta trocar a referência na URL de HTTPS por HTTP e continuar a partir do passo 2 da abordagem anterior. A Figura 4.12 mostra o processo, utilizando as referências.

Perceba que, inicialmente, o atacante recebe as requisições através da porta 80, utilizada pelo HTTP, repassando as mesmas por esta. Entretanto, ao fechar as conexões seguras com o servidor, o atacante irá repassar estas requisições para porta 443 do TLS, usada para conexões seguras. A recíproca também acontece quando o atacante recebe a página completa do servidor e então tem que enviar a mesma para a vítima. Portanto, o atacante age como um *proxy*. Note também que o atacante está fechando uma conexão insegura com a vítima e uma conexão segura com o servidor.

No fim de ambas as abordagens, a vítima terá a página *web* requisitada, sem nenhuma referência a HTTPS, tanto na sua URL, como nas URLs dos *hyperlinks*. As suspeitas explícitas presentes nos navegadores são: a falta da palavra HTTPS no começo da URL da página na barra de endereços, e a ausência de um indicativo de que a sessão está assegurada pelos protocolos SSL/TLS. Ambos, entretanto, são um detalhes sutis nos navegadores e imperceptível para muitos usuários.

Já o atacante terá todas as páginas requisitadas pela vítima, uma vez que foi ele quem fechou as conexões seguras com o servidor e recebeu as páginas de fato.

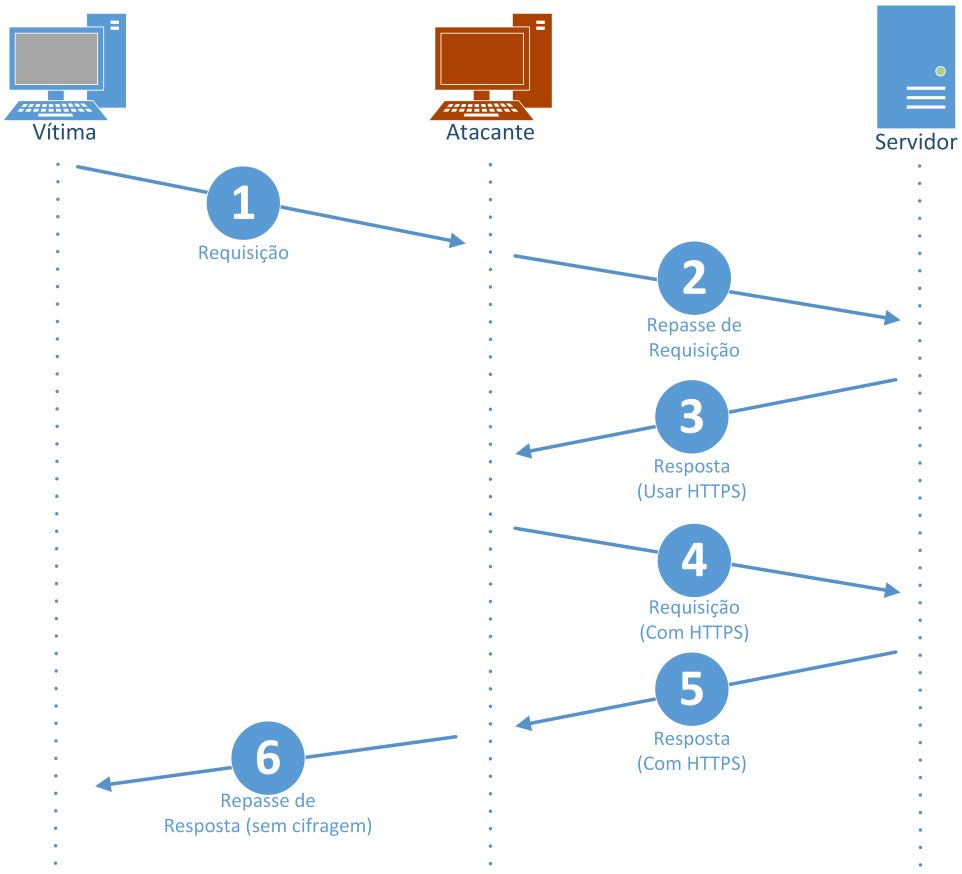


Figura 4.12: Processo de *relay* complexo

4.6 *Relay* Complexo

Transpondo a técnica do SSL *Stripping* para o *relay*, tem-se agora um *overhead* maior. Neste processo, não é mais possível o atacante fazer um repasse simples. Agora ele deverá agir como um *proxy*, lidando com as requisições do cliente para o servidor, fechando a conexão segura com o servidor, obtendo as páginas *web*, modificando-as, retirando as referências ao HTTPS, e, finalmente, enviando à vítima.

Como dito anteriormente, esta técnica cria uma canal de comunicação seguro entre atacante e servidor e um outro canal inseguro entre atacante e vítima. Este último canal abre margem a um problema no ataque: o navegador da vítima não irá sinalizar ao usuário que comunicação está sendo protegida por um algorítmo de cifragem. Este sinalização ocorre, normalmente, de duas formas:

- A URL da página requisitada é iniciado por `http` e não por `https`;
- Um sinal na interface gráfica, normalmente um ícone de um cadeado verde, não é mostrado na barra de endereços do navegador.

A ausência desses dois pontos configura uma assinatura do ataque, mesmo que sutil. Ainda assim, é uma assinatura considerada tolerável.

As Figuras 4.4 e 4.12 ilustram como o ataque vai se dar, de forma que a técnica do SSL *Stripping* é auto suficiente para a realização do *relay* complexo. Concluindo, foi mostrado como obter os dados trafegados entre cliente e vítima em canais seguros por SSL/TLS.

Capítulo 5

Renderizando Telas

Em posse dos dados almejados, o atacante poderá utilizá-los como bem entender. No caso da ferramente Webspy, o objetivo é renderizar as telas *web* localmente, em um navegador à escolha. Para tal, é necessário ter uma compreensão de como é montada uma página *web* dentro de um navegador moderno, levando em conta o leque de opções de navegadores.

Mais que isso, é necessário entender os tipos de dados a serem manipulado, ou seja, os arquivos com os quais um navegador estará apto a receber para renderizar as telas. Além disso, é necessário ter uma visão da implementação da nova *web*, com foco em como suas páginas se comportam e que tipos de propriedades as tornam tão diferentes de implementações anteriores [18] [36].

Neste capítulo será dada a abordagem teórica sobre tais assuntos, listando os problemas encontrados e suas soluções.

5.1 Navegadores

Navegadores são uma das aplicações, atualmente, mais utilizadas em todo o mundo e sendo utilizados em diversas plataformas. Segundo [5], baseiam-se no protocolo HTTP, com suporte ao SSL/TLS, tendo o objetivo de requisitar conteúdo da *web*, a fim de renderizá-lo ao usuário para que haja uma interação entre este e o conteúdo.

A forma de conteúdo das páginas *web* é geralmente apresentada sobre o HTML (*HyperText Markup Language*), definida em [14]. É uma linguagem de marcação para textos para a *web*, largamente utilizada no início da *Internet* e, atualmente, encontra-se na versão 5. Para a exibição de uma página, o navegador necessita minimamente de um documento HTML, o qual será o conteúdo textual básico para exibição. Entretanto, outros elementos podem ser adicionados a este:

- ***Cascading Style Sheet*:** também chamado de **CSS**, é o arquivo que ditará regras de estilo para o documento HTML, modificando posição, cor, tamanho e outras propriedades de elementos do documento;
- **JavaScript:** é uma linguagem de *script* que não necessita propriamente de um pré-processamento. Adiciona dinamismo ao documento HTML, cria uma ponte entre navagador e plataforma cliente, além de permitir requisições de conteúdo de

forma assíncrona, ou seja, sem necessariamente estar vinculada à requisição inicial do documento raíz da página;

- **Mídias:** existem vários tipos de arquivos de mídia, dentre os quais podemos citar: imagens, áudios, vídeos, fontes e outros. Vários tipos são suportados e estes compõem partes específicas do documento.

O Código 5.1 mostra um documento simples com inclusão de outros objetos. Existem outros tipos de arquivos que podem ser inseridos nas páginas, como *servlets* Java e outros, porém o foco serão os três tipos citados acima. É importante relembrar que estes arquivos para serem incluídos no documento HTML devem ser primariamente requisitados, ou seja, deve haver uma referência explicita no documento onde estes devem ser incluídos.

Listing 5.1: Exemplo de código HTML, com referências a outros objetos

```
<html>
  <head>
    <meta charset="UTF-8">
    <title>Pagina de Teste!</title>
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js">
    </script>
    <script type="text/javascript"
src="http://exemplo.com.br/templates/script.js"> </script>
    <link rel="stylesheet" type="text/css"
href="http://exemplo.com.br/templates/style.css">
  </head>

  <body>
    <h1>Psgina de teste Webspy</h1>
    <img src="http://exemplo.com.br/templates/logo.png" />
    <p>Ferramenta Webspy para renderizar telas de possiveis vitimas</p>
  </body>
</html>
```

Identificados os tipos de arquivos com os quais um navegador irá lidar e, consequentemente, a ferramenta Webspy, é necessário entender como um navegador lida com estes arquivos recebidos, entendendo a ordem de renderização e requisição dos mesmos.

5.1.1 Montagem de uma Tela

Todo o embasamento desta seção está em [5] e [40]. Por se tratar de uma aplicação que lida com o HTTP, o navegador, naturalmente, irá lidar com o TCP, ou seja, canais seguros e dedicados serão estabelecidos com o servidor. Inicialmente, um usuário requisita uma página *web*, digitando um determindo endereço na barra de endereços. Aqui, é importante relembrar que o navegador trabalha de forma colaborativa com a *cache*. Ou seja, ao receber objetos, o navegador irá armazenando-os para, possivelmente, reaproveitá-los. Logo, quando um endereço é requisitado, o navegador irá averiguar se possui em sua *cache* a tradução do nome do endereço para o endereço IP desejado. Caso não tenha, terá que fazer uma requisição DNS para este endereço.

Com o endereço em mãos, o navegador agora pode iniciar uma conexão TCP com o servidor desejado para realizar a requisição HTTP e receber uma resposta apropriada. Quando uma requisição HTTP é feita, mais uma vez o *cache* entra em ação: porções do documento solicitado podem estar guardadas localmente, e estas podem ser aproveitadas. Isto configura um problema para o Webspy: à medida que a vítima aproveita sua *cache*, mais informações deixam de ser requisitadas e, consequentemente, nunca chegarão ao atacante. Para tal, deverá ser adotada uma forma de bloquear o uso do *cache* na requisição de páginas.

Quando o cliente requisita uma página, este requisita primariamente um objeto de texto HTML. Desta forma, percebe-se que não há requisição por arquivos CSS, *JavaScript* ou de mídia. Todos os objetos auxiliares que fazem parte daquele documento serão requisitados de forma individual, cada um em uma requisição HTTP. Nesta parte, é interessante notar que estes objetos não necessitam vir do mesmo servidor que entregou o documento HTML. São objetos distintos e, portanto, podem ser requisitados de diferentes servidores. Além disso, podem ser requisitados por diferentes portas e processos. O que o navegador espera é apenas receber o objeto para poder renderizá-lo na tela.

Dentro dos navegadores, diferentes *engines* são utilizadas para a renderização dos objetos recebidos. Não é foco deste trabalho entender como estas funcionam mas, apenas que elas esperam o recebimento dos objetos para então renderizar a tela completa. Uma vez que todos os objetos foram entregues, o navegador fecha as conexões abertas, enviando segmentos TCP com a *flag* `FIN`.

5.2 A Nova *Web*

A primeira versão da ferramenta Webspy apresentava resultados satisfatórios para as primeiras abordagens da *web*: requisições simples, não cifradas, sendo respondidas com páginas estáticas, sem a presença de sessões de *login* de um usuário em uma aplicação *web*. Entretanto, o ambiente atual da Internet mudou drásticamente. Todas as características citadas já não são o padrão e, por isso, interferem na eficácia de ataque da versão inicial do Webspy. Nesta seção, serão abordadas as técnicas atuais que acabaram por comprometer a primeira abordagem do ataque.

5.2.1 Cookies

Cookies são definidos em [17] e [16]. Uma vez que o HTTP não guarda estado de suas conexões, *cookies* acabam sendo uma estratégia para suprir esta demanda. Estes são pequenas quantidades de dados temporários, guardados no cliente e no servidor, de forma a permitir o servidor conseguir identificar que um cliente de conexões anteriores está realizando um novo acesso. Este esquema é baseado em quatro componentes:

- **Cabeçalho da requisição:** a transferência de dados dos *cookies* se dá por meio dos cabeçalhos HTTP. Como mostrado no Capítulo 5, existem cabeçalhos de controle e dados dedicados à *cookies*. Contém os dados que o cliente deseja enviar;
- **Cabeçalho da resposta:** é preenchido os dados que o servidor envia. Muito utilizado para manter dados recorrentes nas aplicações;

- **Arquivo local no cliente:** serve como uma forma de guardar as informações do servidor até uma futura sessão, evitando a perda de informações de sessões já estabelecidos, podendo resgatalá-las;
- **Banco de dados do servidor:** todo servidor pode ter a capacidade de armazenar uma relação de clientes que os visitaram, para dar a correta funcionalidade aos *cookies* estabelecidos. Além de ser guardada a identificação do cliente, o servidor pode armazenar diversas outras informações que desejar.

A Figura 5.1 mostra um exemplo de como um *cookie* é estabelecido e utilizado, por meio da linha de cabeçalho *Set-cookie*, utilizada ao longo das mensagens HTTP. Perceba que, inicialmente, o arquivo de *cookies* do cliente está vazio. Desta forma, servidores conseguem manter o estado de acesso de usuários, criando novas funcionalidades na aplicação *web*, melhorando experiência do usuário, permitindo que o mesmo reinicie um navegador e, mesmo assim, tenha as mesmas telas de sessões passadas.

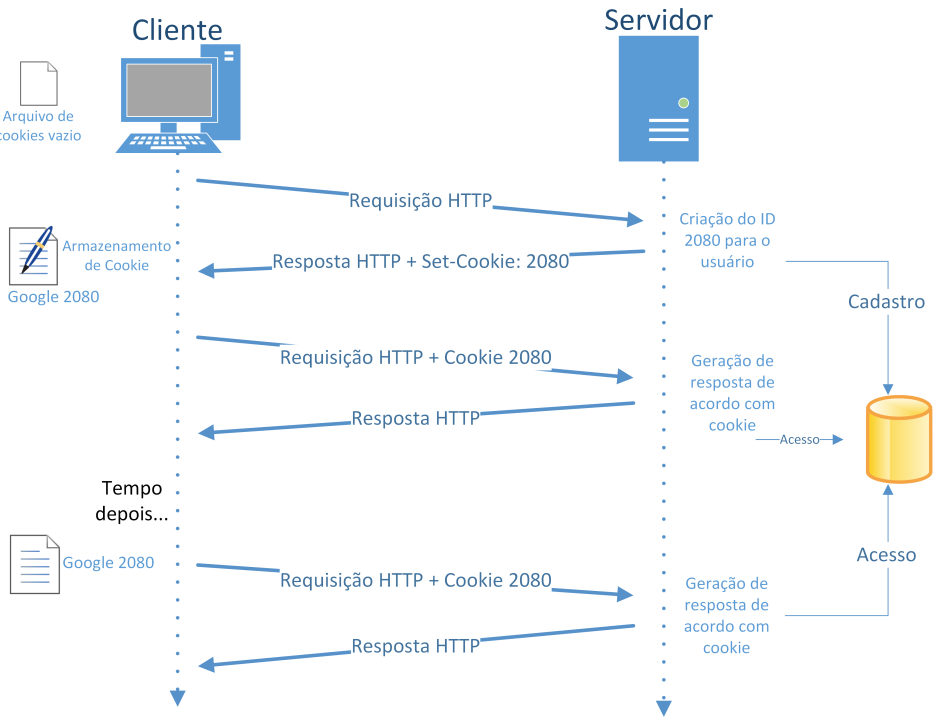


Figura 5.1: Esquema de criação e uso de *cookies* pelo cliente e servidor

Outro fato notório do uso de *cookies* é o fato dos mesmos poderem estar cifrados. Como estes geralmente podem carregar informações importantes e determinantes para ditar o acesso de um dado usuário a um *site* ou aplicaçao *web*, é interessante cifrar os mesmos. Para isto, é possível indicar no cabeçalho HTTP que os mesmos devem estar cifrados e, dessa forma, serem transmitidos apenas por canais seguros, no caso, SSL/TLS.

A importância dos *cookies* na experiência do usuário é determinada pelas ações do servidor. Atualmente, esta influência pode ser alta, com a utilização dos mesmos como forma de manter comportamentos, característicos e determinantes da aplicação ou de forma sutil, onde os históricos de acesso do usuário são utilizados apenas para a elaboração

de sugestões de navegação e outras nuâncias. Entretanto, na *web* atual, isso configura uma característica marcante.

5.2.2 Sessões

Como os *cookies*, sessões são utilizadas para manter estados de conexão, porém o armazenamento dos dados é feito apenas no servidor. A implementação de sessões também ocorre com o uso de *cookies*, entretanto a persistência no cliente é nula pois, caso o *browser* seja fechado, a sessão desaparece. Essa característica dá uma segurança maior à informação armazenada, pois a mesma não irá transitar entre *hosts*.

Na Figura 5.2 há um exemplo de criação de uma sessão, onde um cliente requisita uma página e o servidor envia uma resposta, anexando no cabeçalho HTTP um identificador de sessão dentro de um *cookie*. Este identificador estará salvo de forma consistente apenas no servidor, enquanto que no cliente esta informação é volátil, durando até a próxima requisição. Quando o cliente requisitar uma nova página, desde que o navegador não tenha sido reiniciado, este vai anexar à requisição o identificador recebido, dentro de um *cookie*. Como no servidor, este valor foi salvo, o mesmo poderá manter o estado da conexão, fazendo a associação correta do identificador ao cliente. Todo o processo e manipulações de sessão podem ser encontrados em [3] e [27].

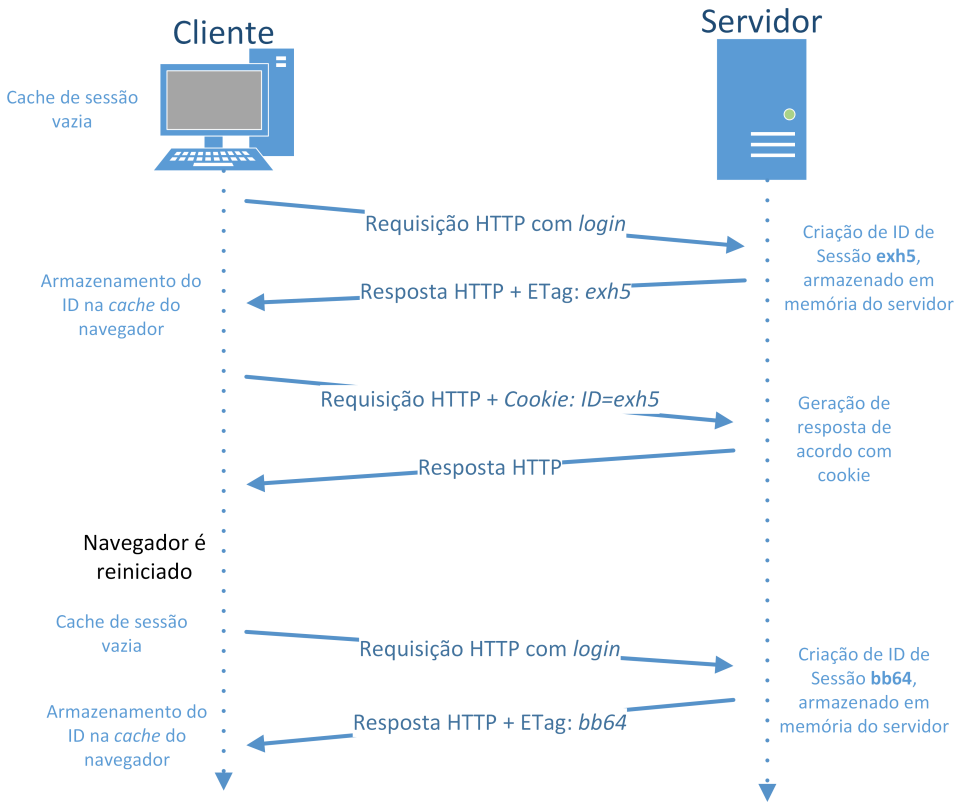


Figura 5.2: Esquema de criação e utilização de identificadores de sessão pelo cliente e servidor

5.2.3 *Javascript* e Requisições Assíncronas

Como já abordado, o *JavaScript* é a linguagem de *script* que trouxe dinamismo e maior interação entre cliente e servidor para a nova *web*. Definido em [9], é voltando primariamente para a *web*, sendo uma evolução do *ECMAScript*. O *JavaScript* acabou por dar uma nova pincelada nos padrões de aplicações na Internet, de forma que grande parte das aplicações desta se baseiam na linguagem.

O *JavaScript* possui um suporte a eventos de usuário e navegador. Estes eventos são apresentados em [1] e estão ligados à ações que o usuário pode ter sobre a página como clicar com o *mouse*, efetuar uma entrada no teclado, deslizar a barra de rolagem da tela, focar em algum elemento da página e diversas outras. Além disso, busca uma integração com o documento HTML por meio de identificadores e classes, que são atribuídos aos nós dos elementos das páginas, usando-os para tomar ações sobre qualquer elemento, alterando suas propriedades e dando mais dinamismo à página.

Também trabalha exaustivamente com a DOM (*Document Object Model*) e a BOM (*Browser Object Model*). Estes dois são estruturas utilizadas pelos navegadores para orientar a organização e estruturação das páginas *web* e propriedades do próprios navegadores. A DOM oferece a árvore de elementos contidos na página HTML retornada pelo servidor, onde cada nó é interpretado como um objeto, contendo diversas informações relativas à seu conteúdo, folha de estilo, hierarquia, dentre outros. Da mesma forma, a BOM oferece uma abstração das propriedades do *browser* em forma de objetos. Este tipo de abstração facilita o *JavaScript* a manipular os mesmos, de modo que ambos são acessíveis por variáveis apropriadas.

Outra característica interessante do *JavaScript* é o poder de realizar requisições independentes para o servidor, ou seja, desatreladas de uma requisição raiz de um documento HTML. Estas requisições são disparadas pelo navegador, podendo aguardar uma resposta do servidor, e também emitem eventos, relativos à espera, estado e recebimento da requisição. Com isso, a linguagem permitiu um comportamento único das aplicações ambientadas na Internet: por diversas vezes, apenas uma página é efetivamente requisitada e todo seu conteúdo é transferido por requisições assíncronas. Este comportamento dá margem a uma imprevisibilidade das requisições de conteúdo e dos seus destinos.

5.3 Problemas e Soluções

Finalmente, é hora de traçar a estratégia de renderização no atacante. Serão listados os problemas e como deve-se lidar com eles.

5.3.1 Suporte à Navegadores

Inicialmente, é interessante notar que o atacante pode renderizar as telas em qualquer navegador local, o que pode variar muito, tendo em vista os diversos navegadores modernos existentes. Estes possuem diferentes estratégias de renderização das telas e suporte à diferentes mídias e *scripts*, porém, todos trabalham sob uma mesma forma de dado: o HTTP. Portanto, o Webspy pressupõe que basta o navegador receber os pacotes HTTP, que este saberá o que fazer com o dado. Evidentemente, deve haver uma ordem na entrega dos mesmos, de acordo com as requisições que o navegador do atacante for produzindo.

5.3.2 Servidor Local e Roteamento de Dados

Outro quesito é a implementação de um servidor local no atacante, ao qual o navegador do mesmo vai recorrer. Esse servidor será implementado dentro do Webspy, recebendo requisições do navegador em uma determinada porta. Ainda nesse ponto, é necessário atentar-se ao alinhamento do navegador com o conteúdo que o Webspy carrega: à medida que a ferramenta interceptar e armazenar os dados sessões *web* da vítima, esta deve também abrir novas telas no navegador, automatizando o processo de renderização de telas.

Ainda, é necessário ter o controle do repasse correto dos dados para cada instância aberta no navegador do atacante. Ao receber dados interceptados de uma conexão, o atacante não possui um processo referenciando a conexão e, portanto, não possui uma indicação direta de qual é o destino correto dos dados recebidos. Por meio das informações disponíveis nos cabeçalhos IP, TCP e HTTP, o atacante deve inferir a qual sessão HTTP um dado pacote HTTP pertence, para que haja o correto retorno das requisições do navegador local do atacante.

5.3.3 Evitando o *Cache*

Como abordado anteriormente, navegadores fazem grande uso do *cache* para diversos objetos. Isto prejudica o ataque, pois a vítima deixa de requisitar objetos que seriam necessários no atacante para obter uma total visualização da experiência do usuário na vítima.

Segundo estudos [13], o uso do *cache* na *web* 2.0 é muito complexo. Graças ao dinamismo introduzido pelo *JavaScript*, versões de *sites* para diferentes plataformas e países e diferentes tipos de acessos para usuários, os conteúdos trazidos por um mesmo servidor variam muito para seus usuários, o que torna o uso de um *cache* consistente muito difícil. Desta forma, apenas uma pequena parcela dos objetos de um documento acabam por serem armazenadas nos *caches* locais dos *hosts*.

Mesmo assim, é necessário lidar com essa barreira. Para tal, é dado um foco nos seguintes campos de cabeçalhos do HTTP, ainda com base em [13]:

- `ETag:` é utilizado para validação de um determinado *cache*. Uma vez que um cliente tem um dado armazenado em *cache* e deve decidir entre requisitar o mesmo novamente ou reaproveitá-lo, este pode confirmar com o servidor qual ação tomar, enviando ao servidor o código deste ETag. O servidor, por sua vez, responde qual a ação a *cache* deve tomar;
- `Cache-Control:` como já mostrado, este campo está presente tanto em cabeçalhos de requisição como de resposta e tem a finalidade de definir como a entidade deve lidar com o uso do *cache* sobre aquele objeto entregue. Dentro deste campo podem vir os seguintes valores:
  - `no-cache`: indica que a resposta entregue não pode ser utilizada para satisfazer solicitações subsequentes com a mesma URL, sem antes verificar com o servidor se o objeto foi alterado. Dessa forma, utilizando o conteúdo do campo `ETag`, o cliente valida se o objeto armazenado em sua *cache* ainda é válido, podendo

reutilizá-lo. Caso contrário, o servidor já terá enviado a última versão do objeto;

- `no-store`: aqui a resposta indica se o objeto não deve ser armazenado sob hipótese nenhuma;
- `max-age`: indica o tempo em segundos que o objeto pode ser mantido na *cache* do cliente;
- `public`: indica que aquele objeto pode ser mantido em *cache* pelo requisitante e demais entidades que receberam a requisição, como *proxys*;
- `particular`: indica que só o destino final pode guardar o objeto no *cache*.

Demonstradas as abordagens presentes neste campo, conclui-se que, para obrigar a vítima a requisitar todos os objetos de um dado conteúdo *web*, sem omissão de requisições por *cache* local, a opção `no-store` deve ser utilizada;

- `Expires:` indica o tempo de expiração de um dado objeto na *cache* do cliente, ou seja, o cliente deve apenas reaproveitar aquele objeto por *cache* até aquele dado momento;
- `Last-Modified:` presente apenas nos campos de resposta, indica o tempo da última modificação daquele objeto;

5.3.4 *Stripping* em Cookies e Mantendo Sessões

Seguindo a linha da nova *web*, o tratamento de *cookies* é uma funcionalidade interessante. O problema encontrado é análogo ao caso do HTTP e HTTPS: alguns *cookies* exigem ser transitados por meio de canais seguros, com SSL/TLS e, dessa forma, por um repasse simples por parte do atacante, os *cookies* viriam cifrados sem a possibilidade do atacante utilizá-los em seu navegador, perdendo a funcionalidade dos *cookies* e, consequentemente, a manutenção de sessões.

Da mesma forma, seguindo a linha do SSL *Stripping*, o *stripping* nos *cookies* também pode ocorrer. Para tal, é necessário que a vítima reinicialize seus *cookies* para que o atacante seja o dono da chave da cifra do *cookie*. A Figura 5.3 ilustra como se dá a técnica, apresentada em [22]. O passo-a-passo pode ser enumerado a seguir:

1. O cliente faz uma requisição a um servidor, já conhecido, enviando seu *cookie* cifrado;
2. O atacante responde a requisição com uma resposta HTTP com código 302, indicando a mesma URL, porém, colocando no cabeçalho da resposta a linha `Set-Cookie: token=deleted; path=/; expires=Thu, 01 Jan 1970 00:00:00 GMT`. Esta linha indica que o *cookie* deve ser excluído, para todos os arquivos da aplicação do servidor, de modo que o mesmo expirou;
3. A vítima agora refaz a requisição, porém sem *cookies*;
4. O atacante repassa a requisição para o servidor. Nesta fase, o servidor provavelmente pode demandar que um canal seguro seja mantido para o envio do *cookie*, sendo necessário estabelecer uma comunicação HTTPS;

5. Ao fim do estabelecimento, o servidor gera o *cookie* usando a chave compartilhada com o atacante e envia a resposta contendo o *cookie*;
6. O atacante agora detém os dados dos *cookies*, que podem ser decifrados e enviados abertamente ao cliente, que irá processar a resposta normalmente.

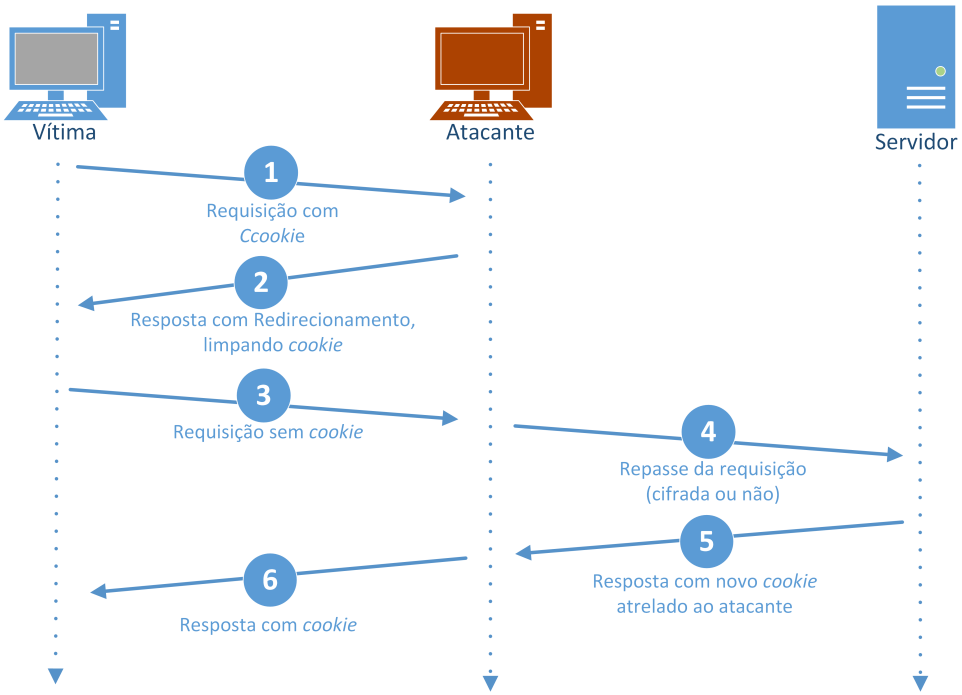


Figura 5.3: Esquematização do *stripping* em *cookies*

Com isso, a ferramenta agora pode ter acesso aos *cookies* da transação e, finalmente, manter sessões. Entretanto, um detalhe não pode ser esquecido: caso o ataque seja realizado no meio de uma sessão, como proceder? Ainda abordado em [22], o estabelecimento de uma sessão (momento de *login*) carrega dados interessantes, como *usernames* e senhas. É possível forçar que uma sessão seja reiniciada, se livrando dos *cookies* entre cliente e servidor, forçando-os a revalidarem a mesma.

Porém, encerrar uma sessão de forma súbita é suspeito, pelo fato das mesmas serem razoavelmente persistentes. Desta forma, o artigo sugere que antes de forçar a revalidação de sessões, o atacante poderia fazer anotações dos *cookies* ativos, de modo a se livrar apenas dos mais antigos, denotando um estado de degradação natural da sessão, não levantando grandes suspeitas. A ferramenta, inicialmente, não trabalhará com esta opção em virtude da necessidade de estudos complementares sobre esta estratégia.

5.3.5 *JavaScript* e o Problema do Evento Perdido

Finalmente, o desafio de se lidar com as requisições assíncronas se mostrou uma das etapas mais difíceis da ferramenta. O *JavaScript* introduziu dois problemas extras ao panorama geral do ataque:

1. Em virtude das requisições assíncronas, mensagens HTTP isoladas são recebidas no atacante, sem algum arquivo base as quais podem ser atreladas. As informações nos

cabeçalhos HTTP destas requisições são muito diretas e escassas e o atacante tem apenas os cabeçalhos de camadas superiores para se guiar. Isto dificulta o roteamento dos dados recebidos dentro das sessões HTTP ativas na aplicação atacante;

2. O *JavaScript* é executado apenas na vítima, criando um ambiente alheio ao atacante. Nesse ambiente, existe uma manipulação e tratamento de dados que não está na visão do atacante, cuminando em uma parcela de dados perdidos ou sem referência para o ataque.

Aqui, encontra-se o problema de diferir uma requisição assíncrona de uma página existente de uma requisição por uma nova página, porém do mesmo *host*. É possível perceber que o problema aumenta a medida que a vítima começa a trabalhar com várias abas de um mesmo *host* (e, consequentemente, mesmo servidor). Logo, mesmo que o atacante consiga definir que uma resposta à uma requisição assíncrona é de um determinado *host*, ele deve diferir a qual aba pertence aquele dado requisitado.

Para resolver tal situação, é necessário apoiar-se na completude dos cabeçalhos HTTP de requisições de novas páginas, as quais, normalmente, trazem mais informações, comparadas a requisições realizadas de modo assíncrono. Além disso, cabeçalhos de requisições HTTP trazem a informação de qual URL raiz o objeto requisitado pertence, no campos *Host* e *Referer*. Dessa forma, é possível criar um esquema de roteamento dentro do Webspy para que os dados sejam corretamente remanejados para as abas corretas.

Por último, um outro problema foi identificado ao longo da implementação do trabalho. Suponha uma página *web* onde, ao usuário clicar sobre um botão com identificador `button`, o mesmo dispara um evento de `mouseClick`, o qual dispara uma requisição assíncrona para o servidor, requisitando uma imagem em *png*. O atacante, naturalmente, intercepta esta requisição e armazena sua referência, esperando a resposta, e repassa-a ao servidor. O servidor prontamente responde a requisição, enviando a imagem em um pacote HTTP, que logo chega ao atacante, que repassa a imagem, porém ainda tem a cópia dela em sua memória. Por fim, suponha ainda que o atacante conseguiu descobrir para qual aba a imagem deve ir. Deste modo, a imagem está pronta para ser enviada para o navegador.

Em primeira instância, o ataque parece ter sido concluído com sucesso. Entretanto, um fato determinante para a renderização não ocorreu: o evento de `mouseClick` no navegador do atacante não ocorreu, logo, quem não requisitou a imagem e, portanto, não espera a mesma, não abrindo nenhuma conexão para recebê-la. Em cima disso, mesmo que o atacante envie a imagem para o navegador, outro problema acontece: o navegador não sabe o que fazer com aquela imagem. Não sabe quem a requisitou e para que objeto da DOM enviar a imagem. Concluindo, a imagem não é renderizada, não por não ter sido interceptada, mas por nunca ter sido requisitada pelo navegador do atacante.

A esta condição se deu o nome de Problema do Evento Perdido, onde, de fato, não há referência daquele evento para que o navegador do atacante possa reproduzi-lo. Isto é uma consequência direta do fato do atacante não ter domínio sobre o que ocorre no cliente. Logo, para resolver tal situação, é necessário inserir um módulo que dê uma situação do cliente para o atacante. A Figura 5.4 ilustra um esquema deste problema.

Para tal, a solução seria a inserção de um arquivo *JavaScript* na vítima que faria uma escuta de eventos atrelada ao identificador do objeto que disparou o evento. Toda vez que tal tupla fosse identificada, a mesma seria enviada ao atacante, juntamente com um identificador de sessão, para ajudar o atacante a rotear a mensagem à aba correta. Recebida a

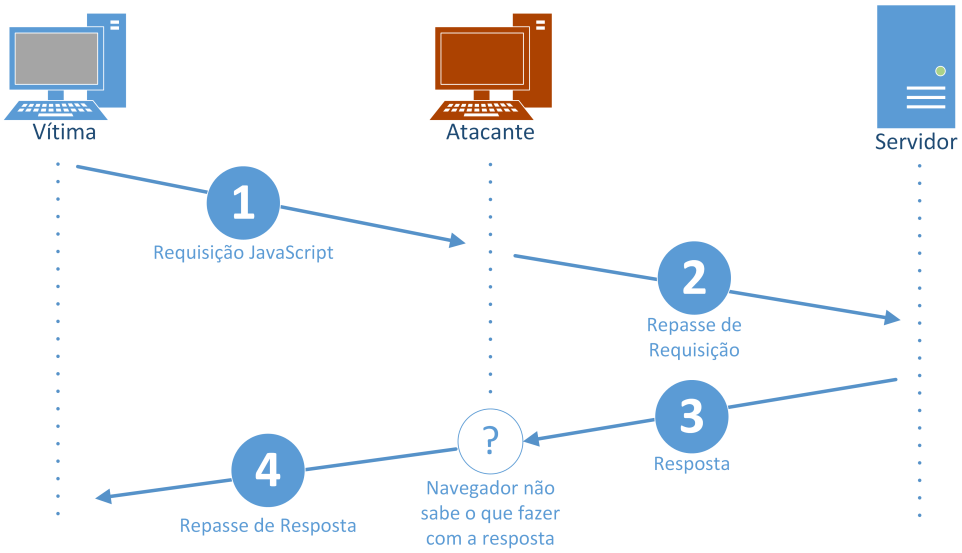


Figura 5.4: Esquematização do Problema do Evento Perdido

mensagem, um outro módulo atacante seria responsável ler a tupla e acionar o evento no elemento indicado pelo identificador. Caso este acionasse uma função ou requisição, a resposta estaria disponível imediatamente ou em breves instantes, dependendo da resposta do servidor condizente com o evento que a vítima disparou. Desta forma, o dado obtido pela requisição assíncrona poderia ser aproveitado pelo navegador do atacante, uma vez que este acionou o evento. A Figura 5.5a solução proposta.

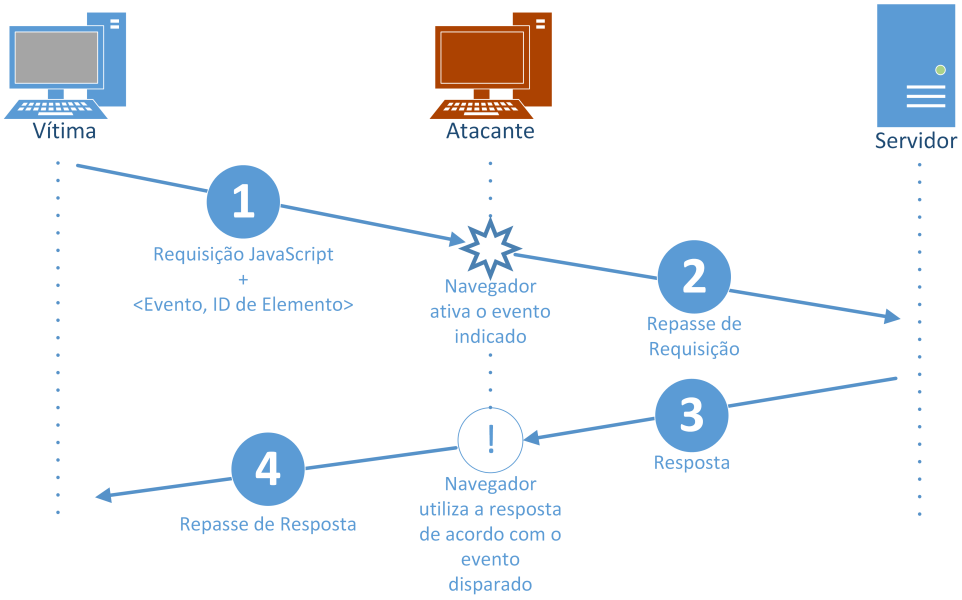


Figura 5.5: Esquematização da solução para o Problema do Evento Perdido, com o arquivo já inserido JavaScript de escuta de eventos já inserido na vítima

Essa estratégia possui um ponto importante: ela acaba por gerar muito tráfico inútil, uma vez que existem vários tipos de eventos a serem considerados onde, em cada um, uma mensagem seria gerada. Entretanto, a mesma que não necessáriamente poderia re-

presentar um evento que iria disparar uma requisição assíncrona ou função do *JavaScript*. Desta maneira, uma melhora na forma de como lidar com este prolema ainda está em aberto, sendo até agora esta a única saída.

Capítulo 6

Decisões de Projeto e Implementação

Finalmente, com toda a base teórica do ataque, pode-se traçar a estratégia prática do mesmo. Neste capítulo serão pormenorizadas as características da implementação da ferramenta, passando pelos problemas encontrados, ferramental utilizado, estratégias adotadas para melhora da performance do programa e resultados obtidos.

6.1 Ferramental

Para este trabalho, a linguagem C++ foi escolhida. Com grande suporte e uma extensa coleção de bibliotecas, esta trouxe facilidades na abstração de entidades do campo de redes de computadores e de instâncias do ataque. Este suporte foi possível graças à orientação a objetos, permitindo a criação de classes que continham as etapas do ataque como responsabilidade. Desta forma, o ataque foi bem modularizada com relativo encapsulamento de classes.

Ainda, herdando as características do C, o C++ dispõe de um livre e poderoso controle da memória, com uso eficiente de ponteiros e funções, garantindo o acesso ótimo aos pacotes recebidos pela interface de rede, além de evitar ponteiros perdidos e vazamento de memória. Além disso, evitava a cópia desnecessária de pacotes recebidos.

Portanto, a estrutura do programa seguiu o que é mostrado na Figura 6.1, onde cada classe será destrinchada ao longo deste capítulo. Paralelamente, algumas bibliotecas interessantes foram utilizadas para evitar a reescrita de código já existe, aproveitando funcionalidades interessantes e métodos já exaustivamente testados.

6.1.1 `libtins`

A `libtins` [20] foi uma das bibliotecas mais importantes deste projeto. A proposta da biblioteca é ser de alto nível, portável e prática, utilizada para a criação e escuta de pacotes de rede, abstraindo protocolos das camadas 2, 3 e 4. Esta possui ainda um ferramental extra, com funções que trazem traduções de endereço de rede para endereço físico, endereço de *gateway* de um *host* e suporte à detecção de interfaces de rede na máquina onde age.

A mesma dá um suporte modularizado a um pacote de rede, abstraindo os protocolos que o mesmo possui e criando uma forma prática de lidar com cada um deles. Feita a abstração, o conteúdo da camada de aplicação, ou seja, de camadas superiores, é tratado

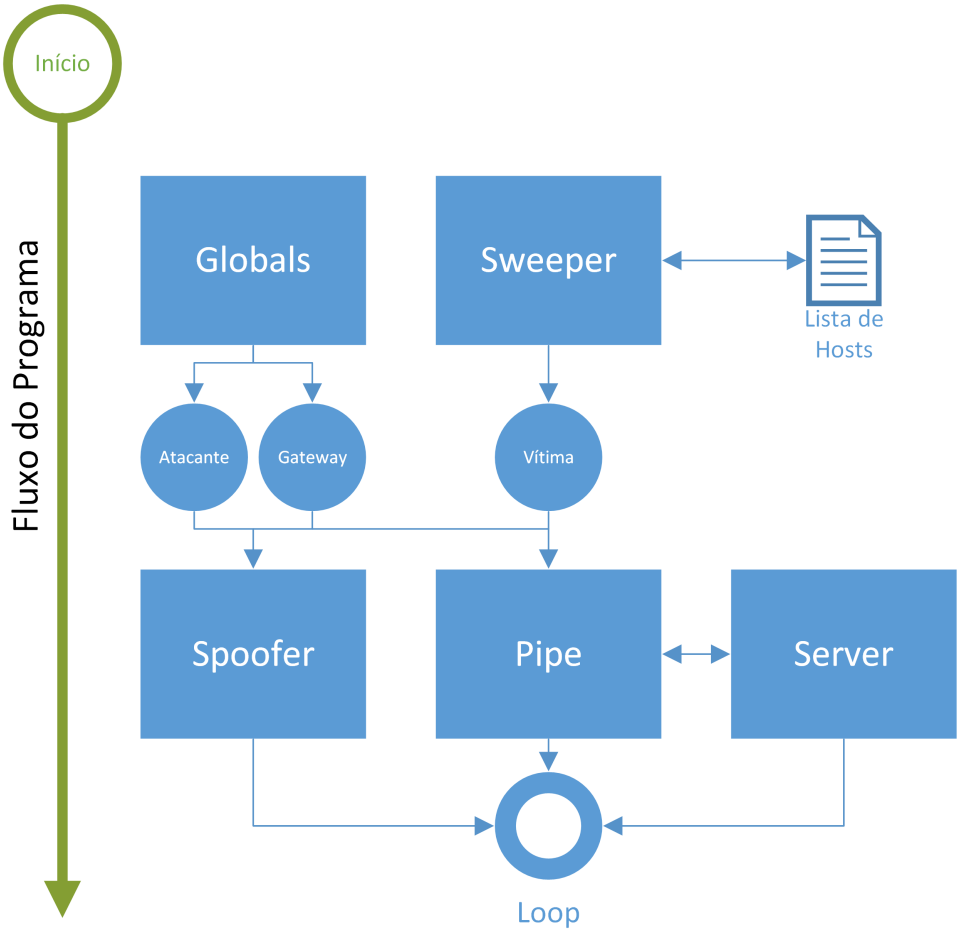


Figura 6.1: Estrutura da ferramenta Webspy, com fluxo de controle. Blocos retangulares são classes e blocos redondos são objetos

como uma *stream* de *bytes*, onde o programador vai lidar propriamente. A Figura 6.2, adaptada de [20], mostra como a biblioteca trata os pacotes.

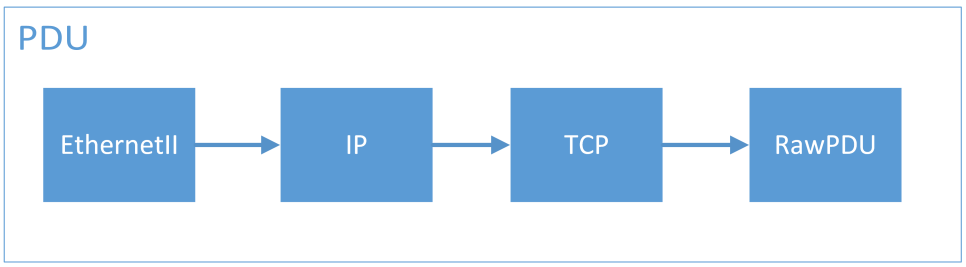


Figura 6.2: Estrutura de um pacote de rede, abtraído pela classe PDU (segmento de transporte), com a estrutura de ponteiro de classes que abstraem os procotolos das camadas da pilha TCP/IP

A biblioteca também possui classes de envio e escuta de pacotes. A classe de envio, `PacketSender`, utiliza uma interface de rede previamente escolhida pelo programador e envia o pacote livremente, por meio de *sockets* próprios para o tipo de nível ao qual o pacote foi manipulado: rede ou enlace. A classe `Sniffer` foi base para a escuta de

pacotes, tendo forte base na clássica `libpcap`. A cada pacote escutado, o mesmo é enviado à aplicação e é tomada uma ação sobre. A mesma também dispunha de um filtro que separava qual tipo de pacote seria enviado à aplicação segundo tipo de protocolos, porta e outros parâmetros.

6.2 Mongoose

Mongoose [2] é uma API de referência para criação de servidores locais em uma aplicação. Sua biblioteca possui métodos que dão suporte à criação destes servidores com diversos parâmetros de configuração, dispondo de um *loop* de escuta de requisições e, através deste, um *handle* para tratamento de eventos. Este último recebe eventos de requisição de conteúdos e toma decisões sobre tais, entregando conteúdo ou não.

Junto com esta classe foi criada toda a inteligência de separação de páginas obtidas durante o ataque, as quais seriam acessíveis ao usuário no atacante.

6.2.1 `pthread`

Biblioteca padrão para criação de *threads* [7]. Foi utilizada para criação das principais *threads* de execução do programa. A Figura 6.3 mostra o esquema das *threads* que foram criadas e utilizadas no programa.

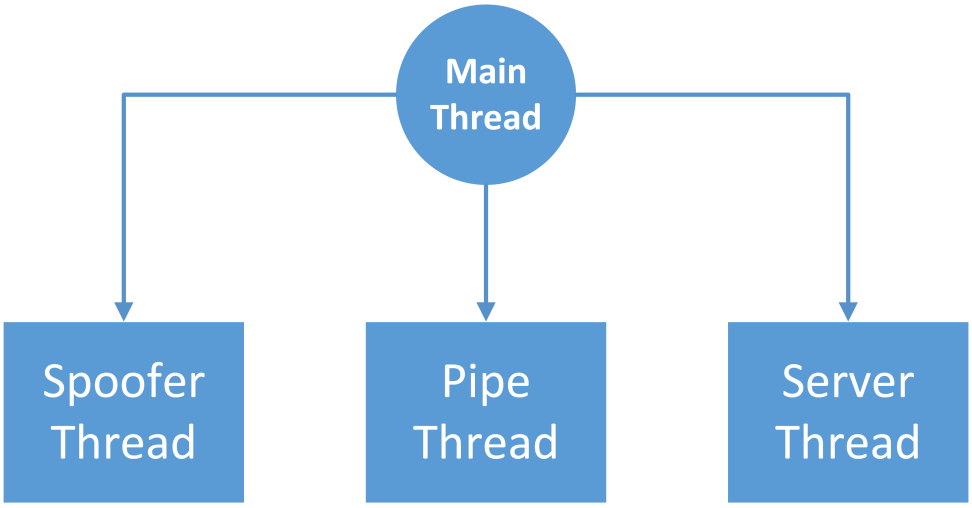


Figura 6.3: Esquema de *threads* na ferramenta Webspy

6.3 Abstrações e Classes de Apoio

Dentro da implementação do Webspy, algumas classes foram criadas para abstrair conceitos do âmbito de redes. Estas são:

- `Host:` abstrai um *host* da rede, tendo como atributos um endereço IP, um endereço MAC e um nome;

- `HTTP:` abstrai uma mensagem HTTP. A classe irá conter métodos de leitura e modificação de cabeçalhos, além de edição do conteúdo da mensagem HTTP;
- `'HTTPSession:` conceito de uma sessão HTTP, criada a partir de uma requisição inicial da vítima por uma página *web*. Cada objeto da classe irá conter a coleção de objetos HTTP representando todos os objetos contidos na página do interceptada;
- `Globals:` classe que contém variáveis e objetos comuns a todas as instâncias do programa, como: vítima, *gateway*, atacante e interface de rede. Além disso, será a classe onde serão feitas algumas configurações de funcionamento da ferramenta.

6.4 Procurando Vítimas

A primeira funcionalidade criada e necessária para a obtenção das informações da vítima se baseou no protocolo ARP, para tradução de endereços de rede e utilização da classe `Host`. A classe criada para este módulo foi a `Sweeper`. Tinha como finalidade entregar um objeto *Host*, que seria a vítima, funcionando sob dois meios: sem ou com parâmetro.

Com parâmetro, a classe recebe um IP, inserido pelo usuário como entrada, correspondendo ao IP da vítima. Recebido este, é feito o seguinte processo: utilizando um método de tradução de endereço de rede para endereço físico da `libtins`, o objeto de *host* pode logo ser montado e devolvido. Sem parâmetro, o pressuposto é de que o usuário deseja receber uma lista de possíveis vítimas para então escolher uma. Dessa forma, a estratégia de ARP *Sweep* entra em jogo.

O envio de *requests* é ditado pela máscara de rede do atacante. Esta é aplicada ao IP base do mesmo, que traz a parte relativa à subrede. Desta forma, é identificada a faixa de IPs à qual podem ser encontradas vítimas. Com essa informação e através de um objeto de pacote ARP, um *loop* modifica os cabeçalhos do protocolo ARP enviando *requests* para cada IP. A *thread* de escuta apenas recebia as escutas e criava os objetos `Host` a partir das respostas do ARP.

Inicialmente, uma *request* era feita e aguardada. Dessa forma, foi necessário ficar atento ao tempo de espera para cada uma destas operações, ou seja, *timeout* de resposta. Uma vez que o atacante poderia fazer uma requisição de um endereço vazio, este nunca iria responder e o atacante deveria então ignorar este após um tempo. Foi necessário identificar qual o *timeout* mínimo ao ponto de todas as ARP *replies* válidas chegarem, equilibrando com o fato do *sweep* em si não demorar muito. *Timeouts* grandes davam um tempo de folga para todas as respostas chegarem, mas criavam um processo muito demorado, enquanto *timeouts* pequenos poderiam resultar na perda de respostas.

Além disso, o processo blocante de aguardar cada requisição resultava em um *sweep* muito lento, agravado em redes sem fio, onde o tempo de resposta das requisições é maior que os de rede cabeada.

Esta questão foi solucionada adotando duas estratégias: uso de *threads* e adoção de um *timeout* global para o processo de *sweep*, parametrizado pelo *range* de IPs. Logo, haveria uma *thread* para envio das requisições, a qual era bem rápida e outra para escuta das respostas. Esta última, permitia que as respostas que chegassem mais rápido fossem atendidas primeiro, acelerando o processo. Para determinar o tempo total do processo, a

seguinte fórmula foi utilizada:

$$T = t_0 + 0.1 \times r$$

Onde T é o tempo total em segundos, t_0 é um tempo inicial fixo de 6 segundos e r é o número de requisições. Dessa forma, o tempo não estava atrelado a cada ARP *request* unicamente. Ao fim deste, o mesmo disparava um sinal, para a *thread* de escuta ser finalizada.

Adicionalmente, esse módulo associa um nome a cada *host* encontrado, referenciado pela coorporação que detém a faixa inicial de *bytes* do endereço MAC de cada *host*. Isto é feito como uma ajuda ao atacante para identificar possíveis vítimas alvos.

6.5 *Spoofing*

Esta seção detalha no ataque propriamente dito, sendo uma das entidades mais importantes do trabalho. Aqui entra a primeira parte do ARP *Spoofing*, o ARP *Poison*, onde as tabelas ARP de cada *host* vítima seriam alteradas em prol do ataque. Nesta classe, é disparada uma das *threads* da aplicação, apelidado de *Thread* de *Spoofing*.

A implementação desta *thread* foi relativamente simples. Tendo todas as entidades prontas por meio da classe `Globals`, a classe continha dois objetos de pacote de rede do tipo ARP, que foram moldados segundo a teoria do ARP *Poison*: um destinado à vítima e outro ao *gateway*. Desta forma, os dois pacotes eram primariamente enviados à cada vítima, de forma a iniciar a entrada falsa nas tabelas de ambas.

A primeira abordagem foi a de criar uma *thread* que reenviava esses pacotes a cada 2 segundos, de forma a manter as entradas falsas nas tabelas das vítimas. Entretanto, esta abordagem se mostrou falha, em algumas situações específicas:

1. Quando o *Poison* foi testado sem o *relay* e a vítima requisitava uma página, esta não recebia resposta e seu navegador emitia a mensagem de falha de conexão com a Internet. Nessa hora, a vítima começava a enviar diversas ARP *requests* com o IP do *gateway* sendo requisitado, como forma de reestabelecer a conexão. Eventualmente, entre os espaço de 2 segundos em que o atacante enviava os *replies* falsos, a vítima acabava por receber um *reply* válido do *gateway* e, dessa forma, recebia páginas que não eram direcionadas primariamente ao atacante. De fato, a situação sem o *relay* não deve ser foco mas, supondo que, com o *relay* implementado, a vítima requisite uma página inválida. Nesta situação, o navegador também irá emitir uma mensagem de falha de conexão e as *replies* serão emitidas. Consequentemente, o ataque poderá ser comprometido por esse comportamento;

2. Em algumas redes, especialmente em redes coorporativas protegidas por *firewall*, alguns *hosts* costumam enviar *replies* válidas para endereços IPs conhecidos por eles na mesma subrede, de modo a revalidar as suas entradas nas tabelas ARPs de tais vizinhos. Desta forma, uma resposta válida acabava chegando na vítima e comprometia o ataque;

3. Em um comportamento natural do protocolo ARP, os prazos de validade das entradas da tabela de um *host* indicam que o mesmo deve refazer a consulta do endereço, lançando uma *request*. Em alguns casos, é comum que a resposta a esta possa ser dada entre o intervalo de 2 segundos após o ataque.

Finalmente, uma segunda e sucessiva estratégia foi executada: foi lançado mão de uma *thread* que escutava todas as mensagens ARP que chegassem ao atacante. Toda vez que esta detectava uma ARP *Request* procurando por uma das vítimas (independente de quem perguntava) ou uma ARP *reply* válida de uma das vítimas (que naturalmente era direcionada ao atacante), eram enviados os pacotes ARP falsos após 0.5 segundos. Assim, a estratégia se pautava na demanda de manter o ataque e não era embasada em uma constância desnecessária, atacando os pontos abordados acima da seguinte maneira:

- Sempre que uma vítima estiver tentando procurar por outra, o atacante envia as *replies* falsas (esperando um pequeno tempo para isso), ou seja, sempre que for notada a procura por uma das vítimas, o ataque é renovado, garantindo que qualquer entrada válida que tenha sido criada seja sobreposta;
- Quando uma vítima está sempre se anunciando à subrede (caso do *firewall*), esta, eventualmente, irá anunciar-se para a outra vítima. Assim, o atacante tenta sobrepor esta resposta válida, enviando a entrada falsa quando detectar a *reply* em *broadcast* da vítima anunciante. Esta estratégia vale para o problema 2 e 3 citados anteriormente.

6.6 *Relay*, *Stripping* e a Obtenção de Dados

Esta parte foi o núcleo da elaboração da ferramenta e, com certeza, a parte mais trabalhosa de todas. Todas as funcionalidades aqui foram inseridas na classe Pipe, o que se mostrou uma sobrecarga de responsabilidades para uma só classe, sendo então muito passível a refatorações futuras para melhor balanceamento de responsabilidades e carga. Além disso, a classe foi a segunda *thread* que o programa executava.

Como na primeira, esta tinha um módulo de escuta de pacotes, desta vez com o filtro `tcp port 80 || tcp port 53 || udp port 53`. A primeira parte do filtro serviu para aceitar todos os pacotes provenientes ou destinados à porta 80 e enviados sob o protocolo TCP. Esta assinatura é típica de pacotes que carregam HTTP, padrão para servidores *web* de todo o mundo. As duas outras partes foram necessárias para o correto *relay* de pacotes que carregavam requisições e respostas DNS, protocolo de tradução de nome de *host* para endereço IP, para que o *host* vítima conseguisse resolver os endereços requisitados. Estes em si não eram importantes para o atacante e ficaram limitados apenas à ação de *relay*.

6.6.1 *Relay*

Por meio do *sniffer* incluído na classe `Pipe`, cada pacote enviado pelas vítimas era enviado a um método de *callback*, o qual se incumbia de verificar se o pacote era físicamente endereçado ao atacante. Em caso positivo, era verificado se o MAC de origem pertencia a vítima ou ao *gateway*, excluindo comunicações de *hosts* alheios ao ataque. Quando confirmado que o pacote participava do ambiente do ataque, os cabeçalhos *Ethernet* eram modificados, como MAC de origem sendo definido como o MAC do atacante e o MAC de destino como o MAC da entidade que deveria receber o pacote.

Caso o pacote tivesse sido enviado pelo *gateway*, era necessário verificar se o IP de destino era igual ao IP da vítima. Esta verificação era necessária para que nenhum pacote

endereçado logicamente ao atacante fosse incluído no ataque, como, por exemplo, pacotes provenientes de sessões *web* no atacante.

Pacotes endereçados logicamente ao atacante que deveriam ser incluídos no processamento do ataque seriam pacotes de conexões HTTPS estabelecidas entre atacante e servidor. Entretanto, a gerências destes pacotes era efetuada pelo módulo de *stripping*.

6.6.2 *Obtenção dos Dados*

A obtenção de dados foi uma tarefa que exigiu grande refatoração. Inicialmente, a `libtins` não estava sendo utilizada e a detecção de tráfego, com seu correto armazenamento, era feita por meio de *buffers*. Porém, a `libtins` oferecia duas classes: `TCPStream`, a qual abstraia uma conexão TCP, acompanhando seu estado, até o fechamento, angariando os dados trafegados através desta, e a `TCPStreamFollower`, que realizava uma junção ideal entre o `Sniffer` e a `TCPStream`, escutando os pacotes trafegados pelo atacante e criando os objetos de mapeamento das conexões TCP.

Estas classes ofereceram a arquitetura perfeita para que o Webspy conseguisse manter simultaneamente *relay*, análise e modificação de tráfego por sessão HTTP. O método de escuta da `TCPStreamFollower` permitia a passagem de dois métodos de *callback*: uma para cada segmento TCP contendo dados recebidos e uma para quando a conexão era finalizada e os pacotes das conexões remontados, tanto do lado servidor como do lado cliente.

Entretanto, foram necessárias algumas adaptações na *libtins*, para que paralelamente a este recebimento, a lógica de ataque de *spoofing* fosse executada, ou seja, que os cabeçalhos *Ethernet* fossem modificados para dar continuidade ao ataque. As funções de *callback* da `TCPStreamFollower` recebiam os pacotes sem os cabeçlhos de camada de enlace e com cabeçalhos incompletos de camada de rede e transporte, impossibilitando a adaptação para o *relay*. Consequentemente, foram necessárias adatapções para que o pacote fosse recebido por completo nos métodos de *callback*.

Adicionalmente, foi necessária realizar a análise de cada segmento TCP contendo dados, para que fosse possível satisfazer alguns detalhes do ataque como o controle de *cache* e *cookies* na vítima e a realização do *stripping*. Para tal, todo segmento TCP com corpo de dados presente e com a *flag* ACK ativa teria de ser analisado antes de ser enviado. Segmentos com as *flags* SYN, FIN, RST ou sem corpo de dados podiam ser enviados sem análise prévia, para evitar maiores *overheads*.

Por fim, o método de *callback*, que entregava os pares de pacotes HTTP remontados à aplicação, foi o responsável por criar objetos HTTP e enviá-los à classe `Server`, onde os dados seriam roteados às suas corretas abstrações de sessões HTTP. Esta implementação exigiu o uso de um *mutex*, uma vez que a área do servidor, que manipulava tais sessões, mostrou-se uma região crítica.

Grandes Segmentos: *TCP Offload Engine*

Durante os testes de ataques, constatou-se um fato estranho no recebimento de pacotes: por diversas vezes, pacotes com tamanho maior que a MTU eram recebidos pelo programa. Inicialmente, imaginou-se que estes seriam pacotes HTTP completos, previamente remontados pelo *kernel* e repassados à aplicação. Porém, após pesquisas, não

foi encontrada nenhuma referência à tal comportamento, tanto em ambiente *Unix* ou *Windows*.

Durante mais pesquisas foi encontrada a resposta para tal comportamento. Definido e explorado em [6], o **TCP *Offload Engine***, ou TOE, é uma estratégia adotada pelas placas de rede para balancear o processamento de pacotes de rede, no que diz respeito ao protocolo TCP/IP. Este balanceamento, feito de *software* para *hardware*, desafoga o processador da máquina, deixando-o disponível para outros processos. Esta tecnologia se mostrou necessária à medida que a velocidade de recebimento de pacotes aumentou a tal ponto a superar a velocidade de processamento dos mesmos. O suporte TOE é o suporte total à implemtação do TCP, sendo que existem placas com menor suporte que apenas realizam o *checksum*.

Nas implementações baseadas em Unix, o TOE é definido em [15], nomeado como *Large Receive Offload*. Nesse caso, porém, não há um suporte completo ao TOE em *hardware*, uma vez que a comunidade tem forte resistência à tecnologia, pois este poderá introduzir problemas com patentes, manutenção, limitações para programadores e não trazer benefícios tão vantajosos quando comparados à boas implementações da pilha TCP/IP. Entretanto, a implementação da pilha TCP/IP do *Unix* traz algumas características do TOE, passando certas partes do processamento de pacotes para a placa de rede.

Dessa forma, pelo fato de haver uma implementação de TOE ou algo que se assemelhe a este, o sistema onde a aplicação foi executada acabou por fazer a junção de certos segmentos, que ultrapassavam o valor da MTU, repassando-os para o *kernel* e, consequentemente, à aplicação.

Na parte de escuta de pacotes, este comportamento não configurou um problema, entretanto, para o envio dos mesmos, ocorreram problemas. As bibliotecas que implementam o envio de pacotes para o enlace não permitem o envio de pacotes maiores que a MTU e também não implementam a fragmentação do TCP. Desta forma, ao tentar realizar o *relay* de pacotes de dados maiores que a MTU, as funções destas bibliotecas resultavam em um erro operacional, não enviando o pacote e, geralmente, acabando em uma excessão em tempo de execução.

Como solução, foi utilizado um filtro da classe `Sniffer` da `libtins` focado no tamanho de pacotes recebidos na escuta. Este filtro foi configurado para o tamanho da MTU, fazendo com que os pacotes recebidos acima do limite da MTU fossem fragmentados pela biblioteca e passados à aplicação.

6.6.3 *Stripping*

A implementação do *stripping* foi auxiliada por uma biblioteca chamada `libcurl` que necessitou do suporte da `libopenssl`. A primeira é uma biblioteca para requisitar páginas por meio de código em C e a segunda é a que dá suporte ao protocolo SSL/TLS, sendo necessária para realizar as requisições em canais seguros.

Sempre que o método de escuta do `Pipe` detectava um segmento com uma resposta HTTP 302 ou 303, o módulo de requisição entrava em ação, extraindo da reposta a URL em questão e requisitando-as. Desta forma a `libcurl` cuidava de toda a requisição e recebimento da página repassando esta para o `Pipe` que, então, enviava a resposta ao cliente.

Adicionalmente, toda resposta HTTP que chegava ao atacante e deveria ser enviada ao cliente, não importando se fosse enviada sob HTTP ou HTTPS, era tratada pelo módulo de *stripping*, que buscava por *links* com referências ao HTTPS e substituia por HTTP. Essa prática ajudava o ataque ao passo que o cliente agora apenas iria requisitar objetos de páginas HTML por meio de canais inseguros, evitando o *overhead* das conexões SSL que deveriam ser feitas pelo atacante.

6.7 Renderizando

A renderização ficou a cargo da classe `Server`, trabalhando em conjunto com a classe `Pipe` e utilizando a biblioteca `libmongoose`. Um vetor de objetos do tipo `HTTPSession` foi implementado como forma de criar abstrações para cada sessão HTTP que a vítima iniciava. A classe `Pipe` angariava os pacotes HTTP, repassando-os à sua sessão HTTP correspondente ou criando uma nova, caso o pacote fosse identificado como o primeiro dentro de uma sessão.

A criação de novas sessões HTTP depende da detecção de um par de requisição e resposta HTTP, onde um objeto de documento HTML fosse criado. A detecção deste é indicada com a ajuda dos campos *Accept* nas requisições e *Content-type*. Quando este contém a *string* `"text/html"`tem-se a indicação do tráfego de um documento HTML. Cada sessão HTTP é identificada pela junção de duas strings: o nome do servidor, presente no cabeçalho *Host* e a URL do objeto inicial requisitado.

Adicionalmente, tais objetos iriam resultar em objetos adicionais, como arquivos CSS, *JavaScript* e mídias. Para descobrir qual a sessão correta deveria ganhar estes objetos, é verificado o campo do cabeçalho *Referer* presentes em mensagens de requisição, o qual é utilizado para ligar o recurso solicitado à página solicitante. Dessa forma, procurando pela URL do *host* em questão na coleção de sessões ativas, era possível rotear o objeto à página correta.

Quando uma sessão era dada por completa, era disparado um método que inicializava uma aba no *browser* do atacante, requisitando uma página para a aplicação, contendo um identificador. Dessa forma, o servidor poderia servir àquela requisição com todos os dados contidos no objeto `HTTPSession` correspondente.

A abordagem às páginas que sofreram SSL *stripping* era transparente: como estas são entregues ao servidor como HTTP puro, não há diferença na implementação no atendimento à estas páginas.

6.8 Demais Dificuldades

A confecção da aplicação Webspy apresentou outros desafios. Estes oscilaram tanto no campo teórico como no prático. Dentre estes temos:

- **Constantes refatorações:** por vezes o código do programa sofreu modificações, criação e remoção de classes, modificação das estratégias e demais estratégias, a fim de encontar uma arquitetura ideal para a aplicação funcionar com bom desempenho. Além disso, a constante busca por bibliotecas que supriam as necessidades da aplicação também consumiu tempo e resoltou em refatorações grandes ao programa.

- **Documentação do SSL:** constatou-se ao longo das pesquisas que o SSL tem uma documentação muito dispersa e pobre. Diversos *blogs* e *sites* coorporativos comentam brevemente sobre o protocolo, mas não trazem um detalhamento a nível acadêmico e não foi encontrada alguma referência oficial sobre o SSL que pormenorizasse sua implementação. Esta defasagem se estendeu às bibliotecas que cuidassem do protocolo, onde apenas a `libcurl` conseguiu dar o suporte necessário;

Capítulo 7

Conclusão

Por fim, conclui-se este trabalho com um breve capítulo sobre as lições, dificuldades e expectativas para o Webspy, sendo importante notar que a ferramenta poderá sofrer modificações futuras para melhorias e adaptações.

O trabalho se prontificou a mostrar toda a base teórica na qual a ferramenta Webspy se apoiou, detalhando conceitos de redes em geral, ataques *man-in-the-middle* através do ARP *Spoofing*, conceitos de obtenção de páginas *web* e manejo da cifragem por SSL/TLS com o SSL *Stripping*.

Adicionalmente, foi explicado como dependência das sub-redes do protocolo ARP causa uma dificuldade em se proteger do ataque de ARP *Spoofing* e, como brechas na implementação do SSL e o desleixo de navegadores em não alertar corretamente a presença de conexões inseguras indevidas, dão margem à implementação do SSL *Stripping*.

Por fim, no âmbito teórico, foi mostrado como algumas adaptações de abordagens antigas possibilitaram que a ferramenta apresentada pudesse tornar-se novamente funcional em implementações atuais de sub-redes. Entretanto, foi notório o fato de que para implementar um ataque de rede, principalmente que atinja camadas de aplicação, é necessário um grande conhecimento dos protocolo da pilha TCP/IP e de nuâncias destes, sendo necessário estar a par das implementações atuais.

Por fim, o trabalho conseguiu mostrar como implementar uma ferramenta que cria um monitoramento em tempo real das sessões *web* de uma vítima em uma dada sub-rede.

7.1 Trabalhos Futuros

Este documento trouxe o passo-a-passo para criar uma aplicação que implemente o monitoramento em tempo real de telas de uma dada vítima em uma sub-rede. Tópicos sobre o tratamento de eventos disparados na vítima e informações trafegadas com ajuda do *JavaScript* foram definidos, porém ainda em aberto. A definição de uma nova abordagem, mais eficiente e que traga menos ruído a sub-rede ficou como um trabalho futuro.

No âmbito da aplicação a implementação se estendeu aos pontos do monitoramento de páginas *web* trafegadas sob HTTP e HTTPS, com o devido controle de *cache* e *cookies* não cifrados. O tratamento de *cookies* cifrados e sessões dependentes destes foi postergado para um trabalho futuro, apesar de serem abordados.

Por fim, detalhes de interface com o usuário, tratamento de exceções e o remonte de sessões de ataque também ficaram como trabalhos futuros.

Referências

[1] JavaScript Tutorial. http://www.w3schools.com/js/. Acessado em: 12 Jun. 2015. 59

[2] Mongoose - most easy to use Web Server on the planet. https://www.cesanta.com/mongoose Acessado em: 15 Jun. 2015. 68

[3] Mozilla Development Network. https://developer.mozilla.org. Acessado em: 12 Jun. 2015. 58

[4] Wi-Fi (WLAN, IEEE 802.11). https://wireshark.org/Wi-Fi Acessado em: 15 Jan. 2015. 26

[5] M. W. Godfrey A. Grosskurth. A Reference Architeture for Web Browsers. *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005 (ICSM'05)*, pages 661 – 664, September 2005. 54, 55

[6] A. Menon, W. Zwaenepoel. Optimizing TCP Receive Performance. https://www.usenix.org/legacy/event/usenix08/tech/full_papers/menon/menon_html/paper.html Acessado em: 22 Jun. de 2015, April 2008. 73

[7] B. Barney. Posix threads programming. https://computing.llnl.gov/tutorials/pthreads/ Acessado em: 15 Jun. 2015. 68

[8] R. Braden. Requirements for Internet Hosts – Communication Layers. https://tools.ietf.org/html/rfc1122 Acessado em: 25 Abr. 2015, October 1989. 36, 38

[9] M. Bynens. JavaScript, a.k.a. Web ECMAScript: Living Standard, August 2015. 59

[10] S. Cheshire and M. Krochmal. Special-Use Domain Names, February 2013. 11

[11] T. Dean. *Network+: Guide to Networks*. Course Technology Press, Boston, MA, United States, 5th edition, 2009. 1, 17

[12] M. Gast. *802.11 Wireless Networks: The Definitive Guide*. O'Reilly, 1th edition, 2002. 20, 21, 22

[13] I. Grigorik. HTTP caching, January 2014. 60

[14] S. Faulkner T. Leithead I. Hickson, R. Berjon. HTML5: A vocabulary and associated API for HTML and XHTML. http://www.w3.org/TR/html5/ Acessado em: 12 de Jun. 2015, October 2005. 54

[15] J. Corbet. Large Receive Offload. http://lwn.net/Articles/243949 Acessado em: 22 Jun. de 2015, August 2007. 73

[16] K. W. Ross J. F. Kurose. *Redes de Computadores: Uma Abordagem Top-Down*. Addison-Wesley Publishing Company, 4th edition, 2009. 1, 3, 6, 7, 9, 10, 20, 28, 36, 42, 56

[17] D. Kristol. HTTP State Management Mechanism. https://www.ietf.org/rfc/rfc2965.txt Acessado em: 12 Jun. 2015, October 2000. 56

[18] A. F. Lourenço. Webspy: um software de monitoramento de rede, 2013. iii, iv, 1, 2, 20, 33, 34, 54

[19] J. Gondim M. Carnut. ARP spoong detection on switched Ethernet networks: A feasibility study. *In Proceedings of the 5th Symposium on Security in Informatics*, 2003. 31

[20] M. Fontanini. libtins: packet crafting and sniffing library. http://libtins.github.io/ Acessado em: 15 Jun. 2015. 66, 67

[21] H. Ding M. Yang, Y. Wang. Design of Win Pcap Based ARP Spoofing Defense System. *2014 Fourth International Conference on Instrumentation and Measurement, Computer, Communication and Control (IMCCC)*, pages 221–225, September 2014. 31, 34

[22] M. Marlinspike. New Tricks For Defeating SSL In Practice. https://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf Acessado em: 30 Abr. 2015, October 2009. 49, 61, 62

[23] S. Tapaswi N. Agrawal, B. Pradeepkumar. Preventing ARP spoofing in WLAN using SHA-512. *2013 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, pages 1–5, December 2013. 31, 34

[24] IEEE Institute of Electrical and Electronics Engineers. IEEE Standard for Ethernet. *IEEE Standards 2012*, December 2012. 8, 9

[25] IEEE Institute of Electrical and Electronics Engineers. IEEE Standard for Informartion Tecnology - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer(PHY) Specifications. *IEEE Standards 2012*, March 2012. 3, 20, 23, 25

[26] Information Sciences Institute University of Southern California. Transmission Control Protocol. https://www.rfc-editor.org/rfc/rfc793.txt Acessado em: 20 Abr. 2015, September 1981. 36

[27] J. Ousterhout. Cookies and Sessions. http://web.stanford.edu/~ouster/cgi-bin/cs142-fall10/lecture.php?topic=cookie, 2010. 58

[28] P. Pandey. Prevention of ARP spoofing: A probe packet based technique. *2013 IEEE 3rd International Advance Computing Conference (IACC)*, pages 147–153, February 2013. 31, 34

[29] D. C. Plummer. An Ethernet Address Resolution Protocol. http://www.rfc-editor.org/rfc/rfc826.txt Acessado em: 21 Set. 2014, November 1982. 13

[30] J. Postel. Internet Protocol (IP). https://www.ietf.org/rfc/rfc791.txt Acessado em: 12 Set. 2014, September 1981. 10

[31] J. Mogul H. Frystyk L. Masinter P. Leach T. Berners-Lee R. Fielding, J. Gettys. Hypertext Transfer Protocol – HTTP/1.1. https://www.ietf.org/rfc/rfc2616.txt Acessado em: 21 Abr. 2015, June 1999. 38, 80

[32] J. Reschke R. Fielding. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. https://tools.ietf.org/html/rfc7231 Acessado em: 21 Abr. 2015, June 2014. 80

[33] E. Rescorla. *Designing and Building Secure Systems SSL and TLS*. Addison-Wesley Professional, 1st edition, 2000. 42, 44

[34] E. Rescorla. HTTP Over TLS. https://tools.ietf.org/html/rfc2818 Acessado em: 22 Abr. 2015, May 2000. 48

[35] T.J. Socolofsky and C.J. Kale. A TCP/IP Tutorial. http://www.rfc-editor.org/rfc/rfc1180.txt Acessado em: 12 Sep. 2014, January 1991. 7

[36] D. Song. Dsniff. http://www.monkey.org/~dugsong/dsniff/ Acessado em: 12 Set. 2014, 2000. iii, iv, 1, 2, 54

[37] Y. Suga. SSL/TLS Status Survey in Japan - Transitioning against the Renegotiation Vulnerability and Short RSA Key Length Problem. *2012 Seventh Asia Joint Conference on Information Security (Asia JCIS)*, pages 17 – 24, August 2012. 47

[38] C. Allen T. Dierks. The TLS Protocol Version 1.0. https://www.ietf.org/rfc/rfc2246.txt Acessado em: 26 Abr. 2015, January 1999. 47

[39] A. Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 3th edition, 2002. 1, 6, 8, 28

[40] W3C. How does the Internet work. http://www.w3.org/wiki/How_does_the_Internet_work Acessado em: 12 de Jun. 2015, March 2014. 55

[41] W. El-Hajj Z. Trabelsi. ARP spoofing: a comparative study for education purposes. *2009 Information Security Curriculum Development Conference*, pages 60–66, March 2009. 31, 34

[42] H. Zimmerman. OSI Reference Model: The ISO Model of Architecture for Open System Interconnection. *IEEE Transactions on Comunications*, 20(3):425 – 432, April 1980. 7

eita

Apêndice A

HTTP: Códigos e Cabeçalhos

A seguir serão mostrados os possíveis parâmetros das mensagens HTTP dentre eles as operações de requisição, códigos de resposta e os possíveis campos de cabeçalhos tanto nas requisições quanto nas respostas. Todas essas entidade são definidas em [31] e [32].

A.1 Operações

O HTTP trabalha com operações e respostas para estas operações. As operações no HTTP são ações que um cliente pode requisitar ao servidor sobre seus arquivos ou arquivos que o cliente deseja submeter ao servidor. Estas ações, definidas em [31], são:

- `GET`: requisita um objeto ao servidor de forma explícita, por meio de uma URL. Este é um dos métodos onde o cliente pode enviar dados ao servidor e, por utilizar o `GET`, os mesmos são embutidos na URL, de forma apropriada ao servidor. Desta forma, o campo de dados da mensagem fica vazio nesta operação;
- `POST`: funciona como o GET, requisitando um objeto do servidor a partir de uma URL. Também pode enviar dados ao servidor, entretanto, toda a passagem de informações é feita apenas no campo de dados da mensagem, sendo enviada de forma mais acobertada;
- `DELETE`: método que permite que o cliente requisite o remoção de um dado arquivo no servidor;
- `PUT`: permite que o cliente envie um arquivo ao servidor;
- `HEAD`: funciona como um `GET`, porém não espera uma resposta com o campo de dados preenchido, apenas os cabeçalhos de resposta;
- `CONNECT`: utilizado para conexão com *proxy*, para tunelamento de dados.

As operações de `GET` e `POST` acabam por ser as mais frequentes, por três motivos: são o principal meio de requisições HTTP por parte dos clientes, podem conter dados interessantes a respeito requisições e são o principal meio do cliente também poder enviar dados ao servidor.

A.2 Códigos de Resposta

No que diz respeito aos códigos de respostas, estes são divididos em categorias, as quais representam situações de respostas, cada uma com variações específicas. Entre elas, existem as seguintes categorias: informações, sucesso, redirecionamento, erro no cliente e erro no servidor. As seções a seguir mostram cada classe com seus principais códigos.

A.2.1 Informações

Normalmente, são usadas em condições experimentais, aguardando respostas de status de servidor.

- **100 Continue:** o servidor recebeu os cabeçalhos de requisição e o cliente já pode prosseguir com a requisição;
- **101 Switching Protocols:** requisição do cliente para troca de protocolo e resposta do servidor informando que acatará a troca;

A.2.2 Sucesso

Respostas que indicam que a requisição do cliente foi recebida, compreendida e processada.

- **200 OK:** resposta padrão de sucesso. Dependendo da requisição, o servidor pode enviar dados na resposta. No `GET` a resposta será o objeto requisitado, no `POST` esta pode ser o objeto ou apenas dados do resultado;
- **201 Created:** a resposta foi atendida e resultou na criação de um elemento no servidor;
- **202 Accepted:** a requisição foi aceita para processamento, mas o mesmo ainda não foi finalizado;
- **204 No Content:** a resposta foi aceita e processada, entretanto não retornou nenhum conteúdo;
- **206 Partial Content:** a resposta foi aceita e processada, porém apenas parte do conteúdo foi entregue, causado por uma limitação imposta pelo servidor ou pelos enviados cabeçalhos do cliente.

A.2.3 Redirecionamento

Este classe de resposta impõe que o cliente tenha que tomar outras ações para completar a requisição.

- **300 Multiple Choices:** indica que o cliente pode tomar vários caminhos para concluir a requisição. Estes são apresentadas dentro da resposta;
- **301 Moved Permanently:** a requisição atual e as futuras deverão ser direcionadas para a URL apresentada na resposta;

- **302 Found:** esta resposta é um padrão industrial, indo contra os padrões estabelecidos pelo IETF. No HTTP 1.0 a resposta era denominada como temporariamente movida, ou seja, um redirecionamento temporário. Navegadores, entretanto, implementavam o código 303. Na especificação do HTTP 1.1, a resposta 303 foi implementada, porém alguns navegadores e aplicações ainda utilizam o comportamento da versão 1.0.
- **303 See Other:** a resposta para esta requisição deverá ser encontrada em outra URL pelo método `GET`, informada no corpo da mensagem;
- **304 Not Modified:** indica que o objeto desejado não foi modificado desde a data indicada no cabeçalho da mensagem;
- **305 Use Proxy:** indica que o recurso está apenas disponível por um *proxy*. Não é bem suportado por alguns *browsers*;
- **307 Temporary Redirect:** indica que a requisição deve ser refeita com o uso de outra URL (informada na mensagem), porém em requisições futuras deve ser usada a URL original;
- **308 Permanent Redirect:** muito semelhante à resposta 302. Entretanto, neste caso, o método da requisição não pode ser alterado. Por exemplo, caso a requisição inicial tenha sido um `POST`, a nova requisição deve ser um `POST`.

A.2.4 Erro no Cliente

Categoria de respostas que indicam um erro na requisição do cliente, permitindo navegadores lidarem com este tipo de problema.

- **400 Bad Request:** não é possível processar a requisição pelo fato da mesma estar mal estruturada, corrompida ou de alguma forma que não satisfaz o protocolo;
- **401 Unauthorized:** requisição válida, porém não deve ser respondida pelo servidor, pois o cliente não tem autorização de acesso ao objeto. Esta resposta, entretanto, é reservada para métodos que trabalham sobre *login* em aplicações *web*;
- **403 Forbidden:** análoga à resposta 401, porém abrangendo qualquer tipo de conteudo;
- **404 Not Found:** a requisição foi processada, porém o objeto requisitado não foi encontrado;
- **405 Method Not Allowed:** o método requisitado, ou seja, a operação do HTTP, não foi permitida no servidor ao cliente;
- **406 Not Acceptable:** quando um servidor não pode gerar os dados requisitados pelo cliente, vindos no cabeçalho;
- **408 Request Timeout:** resposta enviada quando ocorre um *timeout* no servidor, o qual aguardava outra requisição;
- **410 Gone:** indica que o recurso requisitado não está ou não estará disponível;

- **418 I'm a Teapot:** definido em 1998, como um brincadeira de Primeiro de Abril, definindo que o servidor, que é um bule de chá, respondeu corretamente.

A.2.5 Erro no Servidor

Mensagens relativas a erros ocorridos no servidor. Desta forma, o servidor avisa ao cliente que o erro foi originado no servidor, para que a aplicação tome a providência que desejar.

- **500 Internal Server Error:** mensagem de erro genérica, quando um problema qualquer ocorrer no servidor;
- **501 Not Implemented:** o servidor não reconheceu o método requisitado pelo cliente ou então ainda não consegue implementar o mesmo;
- **502 Bad Gateway:** o servidor estava agindo como um *gateway* ou um *proxy* e recebeu uma resposta estranha do servidor destino;
- **503 Service Unavaiable:** o serviço implementado pelo servidor não está disponível, sugerindo manutenção ou apenas uma indisponibilidade temporária;
- **504 Gateway Timeout:** o servidor estava agindo como um *gateway* ou um *proxy* e não recebeu uma resposta do servidor final em seu *timeout* estipulado;
- **505 HTTP Version Not Supported:** o nome é autoexplicativo: o servidor não suporta a versão HTTP enviada pelo cliente.

A.3 Cabeçalhos de Mensagens

Os cabeçalhos das mensagens HTTP são opcionais. Porém, o uso deles é de grande ajuda para os navegadores e servidores *web*.

A.3.1 Cabeçalhos de Requisição

- **Accept:** Content-Types que são aceitáveis para resposta
- **Accept-Charset:** Conjuntos de caracteres aceitáveis
- **Accept-Encoding:** Lista de codificações aceitáveis
- **Accept-Language:** Lista de linguagens humanas aceitáveis para resposta
- **Accept-Datetime:** Versão aceitável no tempo
- **Authorization:** Credenciais de autenticação para autenticação HTTP
- **Cache-Control:** Usado para especificar diretivas que devem ser obedecidas por todos os mecanismos de cache ao longo da cadeia de requisição-resposta
- **Connection:** Opções de controle para a conexão atual e lista de campos de requisição ponto a ponto
- **Cookie:** Um cookie HTTP anteriormente enviada pelo servidor com Set-Cookie

- **Content-Length:** O comprimento do corpo de requisição em octetos (bytes de 8-bits)
- **Content-MD5:** A soma binária MD5 codificada na Base64 do conteúdo do corpo de requisição
- **Content-Type:** O tipo MIME do corpo de requisição (usado com solicitações POST e PUT)
- **Date:** A data e a hora em que a mensagem foi enviada (no formato "HTTP-date", definido pela RFC 7231)
- **Expect:** Indica que determinados comportamentos de servidor são requisitados pelo cliente
- **From:** O endereço de e-mail do usuário que fez a requisição
- **Host:** O nome de domínio do servidor (para hospedagem virtual), e o número da porta TCP na qual o servidor está escutando. O número da porta pode ser omitida se a porta é a porta padrão para o serviço requisitado.
- **If-Match:** Apenas executa a ação se a entidade de cliente fornecido corresponde à mesma entidade no servidor. Usado principalmente para métodos como PUT para atualizar apenas um recurso se ele não foi modificado desde a última atualização feita pelo usuário.
- **If-Modified-Since:** Permite que um 304 Not Modified seja retornado se o conteúdo está inalterado
- **If-None-Match:** Permite que um 304 Not Modified seja retornado se o conteúdo está inalterado
- **If-Range:** Se a entidade mantém-se inalterada, envia-me a(s) parte(s) que estão faltando; caso contrário, envia-me a nova entidade toda
- **If-Unmodified-Since:** Envia a resposta somente se a entidade não foi modificada desde uma hora específica
- **Max-Forwards:** Limita o número de vezes que a mensagem pode ser transmitida através de proxies ou gateways
- **Origin:** Inicia uma solicitação para compartilhamento de recursos de origem cruzada (pede ao servidor um campo de resposta “Access-Control-Allow-Origin")
- **Pragma:** Campos de implementação específica que podem ter vários efeitos em qualquer lugar ao longo da cadeia de requisição-resposta
- **Proxy-Authorization:** Credenciais de autorização para se conectar a um proxy
- **Range:** Solicita apenas parte de uma entidade. Bytes são numeradas a partir do 0
- **Referer:** Este é o endereço da página da Web anterior, a partir do qual um link para a página solicitada no momento foi seguido. (A palavra "referrer"foi escrita incorretamente no RFC bem como na maioria das implementações, ao ponto que tornou de uso padrão e é considerada terminologicamente correta)

- **TE:** As codificações de transferência que o agente de usuário está disposto a aceitar: os mesmos valores para o campo de cabeçalho de resposta Transfer-Encoding podem ser usados, mais o valor "trailers"(relacionado com o método de transferência "em partes") para notificar o servidor que espera receber campos adicionais no trailer após o último pedaço, de tamanho zero
- **User-Agent:** A string do agente do usuário, do agente de usuário
- **Upgrade:** Pede para o servidor atualizar para outro protocolo
- **Via:** Informa o servidor de proxies por meio do qual o pedido foi enviado
- **Warning:** Um warning geral sobre possíveis problemas com o corpo da entidade

A.3.2 Cabeçalhos de Reposta

- **Access-Control-Allow-Origin:** Especifica quais sites podem participar do compartilhamento de recursos de origem cruzada
- **Accept-Patch:** Especifica quais formatos de documentos remendados este servidor suporta
- **Accept-Ranges:** Quais tipos de alcance de conteúdo parcial este servidor suporta
- **Age:** A idade que o objeto está em um cache proxy, em segundos
- **Allow:** Ações válidas para um recurso especificado. Para ser utilizado por um 405 Method not allowed
- **Cache-Control:** Diz a todos os mecanismos de cache do servidor para o cliente se ele pode armazenar esse objeto em cache. É medido em segundos
- **Connection:** Opções de controle para a conexão atual e lista de campos de resposta ponto-a-ponto
- **Content-Disposition:** Uma oportunidade para levantar uma caixa de diálogo "Download File"para um tipo de MIME conhecido com formato binário ou sugerir um nome para conteúdo dinâmico. Citações estão necessariamente com caracteres especiais
- **Content-Encoding:** O tipo de codificação utilizado nos dados
- **Content-Language:** A linguagem natural ou linguagens do público-alvo para o conteúdo envolvido
- **Content-Length:** O comprimento do corpo de resposta em octetos (bytes de 8-bits)
- **Content-Location:** Um local alternativo para os dados retornados
- **Content-MD5:** A soma MD5, binária e codificada na Base 64, do conteúdo da resposta
- **Content-Range:** Onde essa mensagem parcial pertence à mensagem de corpo inteiro

- **Content-Type:** O tipo MIME deste conteúdo
- **Date:** A data e a hora em que a mensagem foi enviada
- **ETag:** Um identificador para uma versão específica de um recurso, muitas vezes, uma message digest
- **Expires:** Dá a data e a hora depois de quando a resposta é considerada travada
- **Last-Modified:** A data da última modificação para o objeto solicitado
- **Link:** Utilizada para expressar uma relação tipificada com outro recurso, onde o tipo de relação é definido pela RFC 5988
- **Location:** Usado em redirecionamento, ou quando um novo recurso foi criado
- **Pragma:** Campos de implementação específica que podem ter vários efeitos em qualquer lugar ao longo da cadeia de requisição-resposta
- **Proxy-Authenticate:** Solicita autenticação para acessar o proxy
- **Public-Key-Pins:** HTTP Public Key Pinning, informa o hash da autenticidade do certificado TLS de um site
- **Refresh:** Usado em redirecionamento, ou quando um novo recurso foi criado. Esta atualização redireciona após 5 segundos
- **Retry-After:** Se uma entidade está temporariamente indisponível, isso instrui o cliente para tentar novamente mais tarde. O valor pode ser um período de tempo determinado (em segundos) ou de um data-HTTP
- **Server:** Um nome para o servidor
- **Set-Cookie:** Um cookie HTTP
- **Status:** Campo de cabeçalho CGI especificando o status da resposta de HTTP. Respostas HTTP normais usam uma "Status-Line"separada em vez disso, definida pela RFC 7230
- **Strict-Transport-Security:** Uma política HSTS que informa o cliente HTTP quanto tempo para armazenar em cache a única política HTTPS e se aplica a sub-domínios
- **Trailer:** O valor do campo geral do Trailer indica que um determinado conjunto de campos de cabeçalho está presente no trailer de uma mensagem codificada com codificação de transferência fragmentada
- **Transfer-Encoding:** A forma de codificação usada para transferir com segurança a entidade para o usuário. Os métodos atualmente definidos são: fragmentada, comprimida, desinflada, gzip, identidade.
- **Upgrade:** Pede ao cliente para atualizar para outro protocolo
- **Vary:** Diz aos proxies downstream como combinar futuros cabeçalhos de requisição para decidir se a resposta em cache pode ser usada em vez de solicitar uma nova do servidor de origem

- **Via:** Informa o cliente de proxies por meio do qual a resposta foi enviada
- **Warning:** Uma advertência geral sobre possíveis problemas com o corpo da entidade.
- **WWW-Authenticate:** Indica o esquema de autenticação que deve ser usado para acessar a entidade requisitada
- **X-Frame-Options:** Proteção clickjacking: deny - nenhum processamento dentro de um quadro, sameorigin - nenhum processamento se a origem incompatibilidade, allow-from - permitir a partir de um local especificado, allowall non-standard - permitem a partir de qualquer local