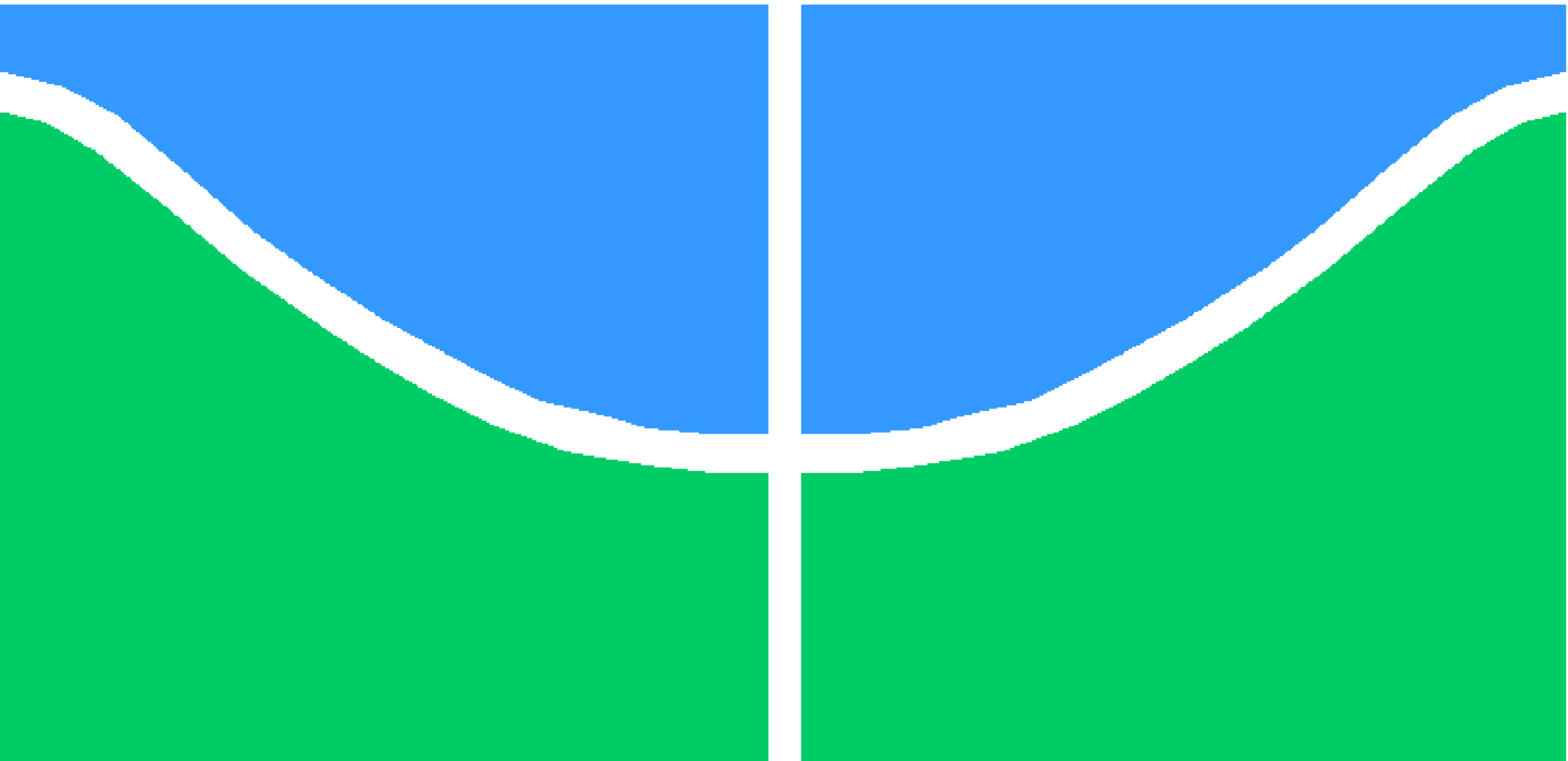

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Engenharia de Software

Automatização de Testes de Penetração em Aplicações Web

Autor: Pedro de Lyra Pereira

Orientador: Prof. Dr. Tiago Alves da Fonseca

Brasília, DF

2018

Pedro de Lyra Pereira

Automatização de Testes de Penetração em Aplicações Web

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Tiago Alves da Fonseca

Brasília, DF

2018

Pedro de Lyra Pereira
Automatização de Testes de Penetração em Aplicações Web/ Pedro de Lyra Pereira. – Brasília, DF, 2018
Orientador: Prof. Dr. Tiago Alves da Fonseca

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2018.

1. SEGURANÇA DE SOFTWARE. 2. TESTE DE PENETRAÇÃO. I. Prof. Dr. Tiago Alves da Fonseca. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Automatização de Testes de Penetração em Aplicações Web

CDU 02:141:005.6

Pedro de Lyra Pereira

Automatização de Testes de Penetração em Aplicações Web

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 12 de julho de 2018 – Data da aprovação do trabalho:

Prof. Dr. Tiago Alves da Fonseca
Orientador

Prof. Dr. Fernando William Cruz
Convidado 1

Prof. MSc. Gustavo Luiz Sandri
Convidado 2

Brasília, DF
2018

Dedico este trabalho à minha família e namorada,
Renata, pelo incentivo e apoio incondicional.

Agradecimentos

Agradeço ao meu professor orientador, Prof. Dr. Tiago Alves, pelo apoio e participação fundamental no desenvolvimento deste trabalho. Agradeço também a todos os professores que contribuíram com o meu aprendizado durante a graduação, em especial ao Prof. Dr. Edson Alves, por sempre estimular seus alunos a cultivarem seus espíritos de campeões.

Resumo

O presente trabalho tem como objetivo o desenvolvimento de uma ferramenta que busca automatizar a execução de testes de penetração em aplicações *web*. Isto posto, foi realizado um estudo sobre a segurança de sistemas de *software*, buscando analisar os principais ataques direcionados a aplicações *web* na atualidade, as técnicas de defesa que podem ser implementadas para proteger uma aplicação destes ataques e formas de avaliar e aprimorar a segurança de um sistema através do uso de testes de penetração. A execução contínua de testes de penetração é uma boa alternativa para avaliar o nível de segurança de um sistema, além de contribuir com a identificação de possíveis vulnerabilidades que podem comprometer a aplicação *web*.

Palavras-chave: Segurança. Aplicação *web*. Teste de penetração. Cross-Site Scripting.

Abstract

The present work has the objective of developing a tool that aims to automate the execution of penetration tests in web applications. A study was made on the security of software systems, analyzing the main attacks directed to web applications today, the defense techniques that can be implemented to protect an application of these attacks and ways to evaluate and improve the security of a system through the use of penetration tests. The continuous execution of penetration tests is a good alternative to evaluate the level of security of a system, besides contributing with the identification of possible vulnerabilities that can compromise the web application.

Key-words: Security. Web application. Penetration test. Cross-Site Scripting.

Lista de ilustrações

Figura 1 – Computadores, roteadores e *switches*. 23
Figura 2 – Ataque *man in the middle*. 28
Figura 3 – Ataque DDoS. 30
Figura 4 – Ataque CSRF. 37
Figura 5 – Representação visual do grafo descrito no Código 2.2. 41
Figura 6 – Grafo que representa um conjunto de páginas *web*. 42
Figura 7 – Arquitetura do projeto. 49
Figura 8 – Desabilitação do *plugin xss_terminate*. 51
Figura 9 – Início do teste. 52
Figura 10 – Execução do teste e submissão de um ataque. 52
Figura 11 – *Log* do servidor do Noosfero. 52
Figura 12 – Vulnerabilidade encontrada. 53
Figura 13 – Resultado do teste. 53
Figura 14 – Código malicioso executado pelo navegador. 53
Figura 15 – Código malicioso embutido no HTML da página infectada. 53
Figura 16 – Início do teste. 54
Figura 17 – Tentativa de ataque XSS. 54
Figura 18 – Identificação de um ataque bem sucedido. 54
Figura 19 – Resultado do teste XSS. 55

Lista de abreviaturas e siglas

ARP	*Adress Resolution Protocol*
CSRF	*Cross-Site Request Forgery*
DoS	*Denial of Service*
FTP	*File Transfer Protocol*
HSTS	*HTTP Strict Transport Security*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
IP	*Internet Protocol*
MAC	*Media Access Control*
RFC	*Request for Comments*
SMTP	*Simple Mail Transfer Protocol*
SQL	*Structured Query Language*
SSH	*Secure Shell*
TCP	*Transmission Control Protocol*
TLS	*Transport Layer Security*
UDP	*User Datagram Protocol*
URL	*Uniform Resource Locator*
XSS	*Cross-Site Scripting*

Sumário

1 INTRODUÇÃO . . . 19
1.1 Contextualização . . . 19
1.2 Justificativa . . . 19
1.3 Objetivos . . . 20
1.3.1 Objetivo Geral . . . 20
1.3.2 Objetivos Específicos . . . 20

2 FUNDAMENTAÇÃO TEÓRICA . . . 21
2.1 Redes de Computadores . . . 21
2.2 Sistemas Web . . . 24
2.3 Segurança de Software . . . 26
2.3.1 Ecossistema e Exemplos de Ataques . . . 27
2.3.2 Segurança na Rede . . . 30
2.3.3 Segurança Interna . . . 33
2.4 Testes de Penetração de Software . . . 38
2.5 Grafos . . . 40
2.5.1 Algoritmos de Travessia . . . 42

3 METODOLOGIA . . . 45

4 RESULTADOS E DISCUSSÕES . . . 51
4.1 Noosfero . . . 51
4.2 Hack.me . . . 54
4.3 Considerações . . . 55

5 CONCLUSÃO . . . 57

REFERÊNCIAS . . . 59

1 Introdução

1.1 Contextualização

A segurança de *software* se tornou um aspecto crítico do projeto de uma aplicação. Um sistema que não implementa uma política de segurança consistente não transmite confiança aos seus usuários. As organizações tendem a investir cada vez mais em negócios *online*, os quais envolvem muitas vezes dados sensíveis de seus usuários, o que torna as aplicações e servidores *web* uma opção atraente aos atacantes (ANSARI, 2015). As falhas de segurança apresentadas por uma aplicação podem ser exploradas por um atacante, ato este que pode comprometer os recursos do sistema, prejudicar seus clientes e acabar com a reputação da organização responsável pelo *software*.

As organizações que desenvolvem aplicações que lidam com dados sensíveis, ou seja, dados que não devem ser divulgados de forma não-autorizada, devem se preocupar com a segurança do seu sistema. Existem diversos métodos que buscam avaliar e aprimorar a segurança de um sistema. Um destes métodos é o teste de penetração, que tem como objetivo simular ataques para testar a segurança de uma aplicação (ANSARI, 2015). A ideia deste tipo de teste é propor que o programador de um sistema adote uma postura ofensiva, na qual se coloca no lugar de um atacante para desenvolver os mecanismos de segurança de um *software*.

1.2 Justificativa

O crescimento de sistemas que utilizam a Internet aumentou a quantidade de dados sigilosos presentes na *web* e como consequência aumentou também o número de interessados em captar estes dados. Além disso, é comum a presença de falhas de segurança em vários sistemas de *software*, pois a engenharia de segurança é negligenciada muitas vezes pelas equipes de desenvolvimento. Essa negligência é fruto do custo adicional relacionado à implementação destes testes e à falta de conhecimento dos programadores sobre segurança de *software*.

A automatização de alguns testes de penetração pode reduzir os custos de implementação relacionados à averiguação da segurança de um sistema e pode facilitar a execução destes testes por desenvolvedores que são leigos no assunto. A execução destes testes contribui com a auditoria de segurança de um *software*, tornando-o mais confiável e comprovando seu nível de resiliência, características desejáveis para garantir a credibilidade do sistema.

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo deste trabalho é o desenvolvimento de uma ferramenta que realiza a verificação de alguns aspectos de segurança presentes em uma aplicação *web* para automatizar a detecção de possíveis falhas de segurança.

1.3.2 Objetivos Específicos

- Realizar um estudo sobre a segurança de sistemas *web*.
- Identificar quais são os principais ataques existentes na atualidade.
- Determinar como os testes de penetração podem contribuir para a evolução da segurança de um sistema.
- Automatizar a execução de uma classe específica de testes de penetração.

2 Fundamentação Teórica

2.1 Redes de Computadores

Ao longo do progresso tecnológico dos sistemas computacionais, pensou-se ser interessante desenvolver algum tipo de mecanismo que permitisse a troca de informações entre computadores, ou seja, implementar um conjunto de tecnologias que viabilizasse a comunicação entre sistemas computacionais. Foi a partir dessa ideia que as redes de computadores passaram a ser criadas. Uma rede de computadores consiste em um sistema formado por um conjunto interconectado de computadores capazes de partilhar informações entre si (TANENBAUM, 2014). A Internet é uma rede global de computadores e pode ser considerada o maior sistema já criado pela humanidade, com centenas de milhões de computadores conectados (KUROSE, 2013).

A Internet pode ser descrita como um conjunto de componentes de *hardware* e *software* operando em conjunto para implementar a transmissão de dados entre os sistemas computacionais espalhados pelo mundo, isto é, uma infraestrutura complexa que provê serviços de comunicação às aplicações de rede[1] (KUROSE, 2013). Os elementos que a formam incluem *laptops*, *desktops*, *tablets*, *smartphones*, *smart tvs*, entre outros dispositivos utilizados pelo usuários para se conectarem à rede. Todos esses dispositivos podem ser considerados computadores, alguns de propósito geral, outros de propósito específico, mas computadores que compartilham uma característica: todos possuem aplicações sendo executadas em seu interior que exigem conexão à Internet para possibilitar seu funcionamento.

As aplicações de rede demandam acesso à Internet porque parte dos dados essenciais para suas atividades não se encontram no computador local em que são executadas. Por exemplo, quando um usuário acessa uma página *web* através de um navegador, o conteúdo desta página deve ser recuperado em algum lugar da rede (o computador onde a página está armazenada). Essa busca ativa uma cadeia de eventos de comunicação que garante que a página seja transmitida para o computador do usuário para que possa ser exibida pelo navegador. Além dos dispositivos utilizados pelos usuários para se conectarem à Internet, existem componentes internos da rede desenvolvidos para garantir seu funcionamento.

A Internet é uma rede que se baseia na comutação de pacotes. A comutação de pacotes é um paradigma de comunicação no qual a informação a ser transmitida é sub-

[1] Uma aplicação de rede é um *software* que faz uso de uma rede para se comunicar com programas sendo executados em outros computadores.

dividida em blocos de dados menores, denominados pacotes. Esses pacotes são enviados individualmente a partir de um emissor com destino a um receptor e percorrem um caminho formado por uma série de enlaces de comunicação[2], possivelmente partilhados por outros nós da rede, até alcançarem seu destinatário. Durante o trajeto, diversos comutadores de pacotes, como roteadores e *switches*, (KUROSE, 2013) se responsabilizam por atuar como um agente intermediário que recebe pacotes em uma de suas interfaces de rede de entrada e os encaminha a uma de suas interfaces de saída de modo a direcioná-los ao seu destino.

A arquitetura de uma rede de computadores determina a forma na qual sua infraestrutura será organizada e implementada. A Internet possui uma arquitetura dividida em múltiplas camadas (aplicação, transporte, rede, enlace e física). Uma arquitetura baseada em camadas permite a análise e estudo de uma parte específica e bem definida de um sistema grande e complexo (KUROSE, 2013). Esse tipo de projeto possibilita a modularização da implementação dos serviços que uma rede provê aos computadores conectados à ela. Cada uma das camadas se responsabiliza em implementar um conjunto de serviços específicos que são necessários para o funcionamento da rede, ou seja, cada camada oferece a sua contribuição para viabilizar a transmissão de pacotes na Internet. Essas camadas formam uma pilha e o fluxo de pacotes é vertical, isto é, apenas camadas adjacentes comunicam-se entre si.

As camadas da rede utilizam protocolos de comunicação responsáveis por fornecer um conjunto de regras que especificam a estrutura, semântica e sincronização das mensagens compartilhadas por duas entidades que pretendem se comunicar. Como cada camada executa um protocolo específico, essa arquitetura forma uma pilha de protocolos. O fluxo de informação nessa pilha é vertical, isto é, apenas camadas adjacentes comunicam-se entre si.

Os protocolos de cada camada adicionam um cabeçalho com informações de controle aos pacotes criados. Alguns elementos da rede, como os roteadores, não implementam todas as camadas, pois suas funções não demandam a utilização dos protocolos oferecidos por algumas delas. A Figura 1 apresenta alguns dos elementos que compõem uma rede e o conjunto de camadas que cada um implementa, além de ilustrar o encapsulamento feito pelos protocolos de cada camada ao envolver seus pacotes com um cabeçalho que contém informações de controle utilizadas em certas atividades administrativas necessárias para o funcionamento do protocolo.

[2] Um enlace de comunicação é um canal de comunicação formado por alguma tecnologia que viabiliza a transmissão de dados entre dois computadores adjacentes.

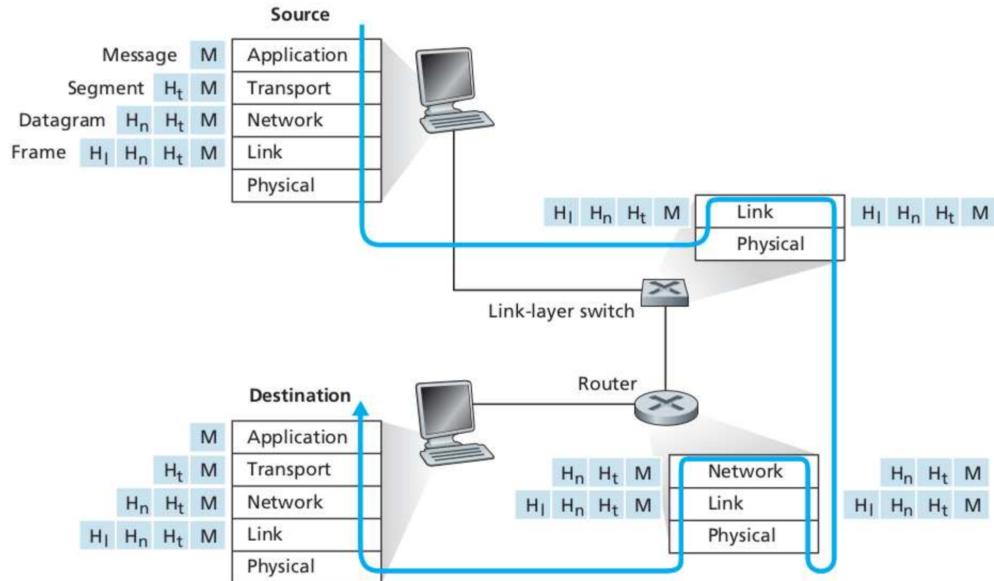

Figura 1 – Computadores, roteadores e *switches*.

Os protocolos utilizados por cada camada provêem serviços específicos às camadas superiores. Toda a atividade na Internet que envolve a comunicação entre duas ou mais entidades remotas é governada por um protocolo (KUROSE, 2013). Segue abaixo uma breve descrição dessas camadas e de alguns dos seus principais protocolos:

- Camada de Aplicação: responsável por determinar a forma em que duas aplicações de rede irão se comunicar (KUROSE, 2013). Esta camada inclui diversos protocolos, como o FTP (*File Transfer Protocol*) para transferência de arquivos, o SMTP (*Simple Mail Transfer Protocol*) para transferência de e-mails, o SSH (*Secure Shell*) para acesso seguro à um terminal remoto. O protocolo que reside nesta camada mais relacionado à este trabalho é o protocolo HTTP (*Hypertext Transfer Protocol*), responsável por implementar a requisição e a transferência de documentos *web*.

- Camada de Transporte: responsável por implementar a comunicação lógica entre dois computadores (KUROSE, 2013) e fornecer uma interface genérica de comunicação para a camada de aplicação. Os principais protocolos desta camada são o TCP (*Transmission Control Protocol*) e o UDP (*User Datagram Protocol*). Cada um destes protocolos fornece seus próprios serviços e as aplicações devem utilizá-los conforme as exigências do seu projeto. A implementação destes protocolos é feita pelo sistema operacional. O sistema operacional mantém um número interno, denominado porta, que identifica unicamente um serviço de rede em execução. Esta porta é utilizada por outras aplicações para especificar ao sistema operacional a qual de seus serviços de rede deseja-se enviar uma mensagem. Por exemplo, a porta padrão de um servidor HTTP é a porta de número 80.

- Camada de Rede: responsável por realizar o roteamento e a comutação dos pacotes de uma rede (KUROSE, 2013). Esta camada se dedica em implementar um algoritmo de roteamento para calcular a rota que um pacote deve tomar para alcançar seu destino. O protocolo de comunicação mais notável desta camada é o protocolo IP (*Internet Protocol*). O protocolo IP utiliza um número de 32 bits denominado endereço IP para identificar uma interface de rede conectada à Internet.

- Camada de Enlace: responsável por controlar a transmissão de pacotes de um enlace de comunicação individual (KUROSE, 2013), ou seja, esta camada se concentra em administrar a transmissão de pacotes entre dois computadores adjacentes. Os protocolos desta camada utilizam um número de 48 *bits* denominado endereço MAC (*Media Access Control*) para identificar uma interface conectada à uma rede local.

- Camada Física: responsável pela transmissão de *bits* individuais em um enlace de dados (KUROSE, 2013). Os protocolos desta camada dependem fortemente da mídia de transmissão que conecta dois computadores adjacentes (cabo coaxial, fibra ótica, atmosfera, entre outros).

2.2 Sistemas Web

Até o início dos anos 1990, a Internet era utilizada principalmente por pesquisadores para a transferência de arquivos acadêmicos (KUROSE, 2013). Sua evolução possibilitou a formação da *world wide web*, também chamada de *web*. A *web* consiste em todo o acervo de documentos e recursos que são acessíveis através da Internet. Esses documentos são identificados por URLs (*Uniform Resource Locator*), que são endereços que localizam um recurso da *web*. Os navegadores são programas que desempenham o papel de recuperar documentos *web* e exibi-los aos usuários. Uma página *web* é um documento de texto escrito na linguagem de marcação HTML (*Hypertext Markup Language*) e é um dos tipos de documento mais comuns que um navegador acessa. O HTML fornece uma notação baseada em *tags* que permite descrever os objetos que compõem uma página *web*, incluindo imagens, formulários, botões, vídeos, textos, *links* que referenciam outros objetos, etc. O navegador interpreta o documento HTML para exibir os elementos gráficos da página ao usuário.

Essas páginas são armazenadas em outro computador, denominado servidor. Um servidor possui um programa em seu interior chamado *web server* cuja função é atender requisições por documentos feitas por um cliente e servir os objetos solicitados. Tanto o navegador quanto o *web server* são aplicações de rede e o modelo em que ambos operam (a forma como interagem) é denominado arquitetura cliente-servidor, onde há uma entidade que permanece sempre *online*, chamada servidor, que aguarda e atende requisições feitas

sob demanda por outras entidades, chamadas clientes (KUROSE, 2013). Nesse contexto, o navegador atua como cliente e o *web server* como servidor.

Um conjunto de transações envolvendo requisições e respostas enviadas respectivamente por clientes e servidores configura uma comunicação e, portanto, exige um protocolo que especifique como estas aplicações irão se comunicar. O protocolo de comunicação HTTP é a essência da *web* e é implementado por clientes e servidores. A função do HTTP é formalizar o modo em que o cliente e o servidor irão se comunicar para permitir a requisição e a transferência de páginas *web*. O HTTP define a estrutura das mensagens trocadas na comunicação entre um cliente e um servidor (KUROSE, 2013) e utiliza o TCP como seu protocolo de transporte subjacente (KUROSE, 2013). Um cliente HTTP que deseja se comunicar com um servidor deve antes iniciar uma conexão TCP. Quando esta conexão é estabelecida, o cliente e o servidor passam a trocar mensagens HTTP.

Os sistemas *web* são aplicações executadas em um servidor que processam requisições feitas por clientes e retornam uma resposta com o objeto solicitado. No início da *web*, estes objetos consistiam em documentos estáticos, portanto, as aplicações atuavam de certa forma como repositórios de páginas *web* e o fluxo de informação era unidirecional (STUTTARD, 2011). Atualmente, a maioria das aplicações *web* são altamente funcionais e implementam, por exemplo, registro e *login* de usuários, transações financeiras, pesquisa e criação de conteúdo por usuários (STUTTARD, 2011). A maioria destas funcionalidades exige o armazenamento de diversos dados envolvidos no contexto da aplicação e, por isso, esses sistemas precisam de um banco de dados para armazenar informações cruciais para seu funcionamento.

As aplicações *web* foram criadas para desempenhar praticamente todas as funções úteis que poderiam ser implementadas *online*. Diversos fatores técnicos contribuíram para a evolução e adoção massiva das aplicações *web*. O protocolo de comunicação utilizado por aplicações *web*, HTTP, é leve e livre de estados[3]. Esta propriedade fornece resiliência em casos de erro de comunicação e evita a necessidade do servidor manter uma conexão aberta para cada usuário (STUTTARD, 2011).

Os principais sistemas operacionais do mercado já vem com algum navegador pré-instalado, o que facilita o acesso do usuário à *web*. Além disso, os navegadores evoluíram muito ao longo dos anos e se tornaram altamente funcionais, sendo capazes de implementar interfaces gráficas ricas e complexas para satisfazer os usuários. Sob a perspectiva do desenvolvedor de aplicações, cada vez mais tecnologias e ferramentas de qualidade são criadas para auxiliar o desenvolvimento *web*, o que facilita o trabalho de programar sistemas *web* e o torna mais produtivo. Todos esses fatores tendem a contribuir com o crescimento da *web*.

[3] Um protocolo é dito “livre de estados” quando interpreta cada par de requisição e resposta como uma transação independente. Ele não mantêm informações sobre requisições passadas.

2.3 Segurança de Software

Os *softwares* podem exigir um certo controle de segurança conforme os requisitos do seu projeto. Esse controle envolve a implementação de diversos mecanismos de defesa que buscam proteger o *software* de acessos não autorizados aos seus recursos. A área de conhecimento responsável pela aplicação de princípios e técnicas que auxiliam a implementação desses mecanismos é denominada Segurança da Informação. A segurança de *software* é a área de estudos que se concentra na compreensão dos riscos de segurança decorrentes do uso de *software* e em formas de gerenciá-los (MCGRAW, 2001), ou seja, os programadores de um projeto devem ser capazes de identificar possíveis vulnerabilidades que um *software* possa vir a apresentar e utilizar técnicas de programação defensiva na implementação de soluções computacionais que buscam mitigar estas falhas de segurança.

Contudo, a programação de mecanismos de defesa que visam fortalecer o sistema exige um amplo conhecimento na área de segurança. Além disso, a implementação dessas camadas de defesa não garante a segurança do *software*, pois ele ainda pode estar vulnerável a algum ataque que a equipe não levou em consideração. O fato é que não existe um *software* totalmente seguro. Isso implica que o desenvolvimento de um sistema seguro trata-se de um processo de evolução contínua e altamente adaptativo à aparição de riscos até então desconhecidos, sendo uma consequência da dinamicidade da área de segurança da informação.

Talvez seja intuitivo imaginar que não valha a pena se preocupar com a segurança de um sistema, já que mesmo que uma equipe se preocupe com sua segurança, ele ainda estará suscetível a apresentar potenciais vulnerabilidades desconhecidas até então. Os *softwares* podem apresentar níveis de segurança distintos que estão associados à qualidade dos seus mecanismos de defesa. Não existe *software* capaz de se defender de todos os ataques que existem ou que possam vir a existir, mas existem *softwares* capazes de reagir bem à ataques conhecidos. Isso garante um determinado nível de segurança para o produto desenvolvido por uma organização. Por fim, a ausência de mecanismos de segurança em um sistema é uma garantia de que ele é inseguro, enquanto a presença destes mecanismos concede aos desenvolvedores o privilégio da dúvida (MCGRAW, 2001).

A crescente conectividade de computadores à Internet aumentou tanto a quantidade de vetores de ataque quanto a facilidade com que um ataque pode ser feito (MCGRAW, 2001). Os sistemas conectados à Internet estão sujeitos a ataques que podem se originar de qualquer lugar do mundo. Além disso, outra tendência que afeta a segurança de *software* é o crescimento desenfreado do tamanho e da complexidade dos sistemas de informação (MCGRAW, 2001). Este fato é justificado por estudos realizados por Blakley, McDermott e Geer (2001) que demonstram a existência de uma relação diretamente proporcional entre o tamanho do código de um *software* e sua quantidade de defeitos. Todos estes fatores indicam a necessidade de se levar em consideração a segu-

rança do *software* durante sua fase de desenvolvimento.

A segurança de sistemas computacionais e de redes de computadores se tornou limitada pela qualidade e segurança dos *softwares* sendo executados em seu interior (MCGRAW, 2001). O aspecto central e crítico em segurança de computadores é o *software* (MCGRAW, 2001). São as vulnerabilidades de um *software*, formadas por uma coleção de erros de implementação e falhas de projeto, que comprometem um sistema computacional. Os computadores são invadidos através da exploração dessas vulnerabilidades. Mesmo assim, a existência de falhas de segurança é comum na maioria dos *softwares*, pois a engenharia de segurança é muitas vezes negligenciada pelas equipes de desenvolvimento devido ao seu custo adicional ao projeto e à falta de conhecimento dos programadores sobre o assunto (WHEELER, 2015). A segurança de um *software* e do sistema computacional em que o mesmo está inserido estão altamente atreladas, pois o comprometimento de um dos dois pode prejudicar o funcionamento de ambos.

2.3.1 Ecossistema e Exemplos de Ataques

Um *software* é executado em um contexto específico formado por um computador, um sistema operacional e, no caso de aplicações de rede, uma rede de comunicações. Todos os elementos deste contexto são essenciais para a manutenção da segurança deste *software*. A presença de uma vulnerabilidade em um destes elementos pode comprometer todos os outros. Um sistema operacional executando um serviço *ssh* que permite o *login* de usuário *root*[4] e, além disso, apresente uma senha padrão, pode comprometer qualquer um dos programas que esteja executando, independente dos mecanismos de segurança internos desses programas.

Antes de analisar a segurança de um *software*, é necessário compreender o ecossistema em que o mesmo está inserido. No caso citado, um ecossistema extremamente hostil devido a uma vulnerabilidade apresentada pelo sistema operacional. Nesse cenário hipotético, um atacante poderia explorar essa vulnerabilidade para acessar o sistema operacional e tomar controle total do computador em que uma aplicação é executada, fazendo com que a mesma possa ser manipulada ou adulterada pelo atacante. Com um raciocínio similar, é fácil compreender que falhas de segurança do computador e da rede que a aplicação pertence também a afetam.

Existe um tipo de ataque chamado *man in the middle* (Fig. 2) onde um atacante se posiciona como um agente intermediário da comunicação entre duas vítimas, sem que elas percebam, para interceptar suas mensagens (STUTTARD, 2011). Essas mensagens podem conter informações sensíveis que não poderiam ser divulgadas para pessoas não

[4] Um usuário *root* é um termo usado em sistemas *Unix-like* para denominar usuários de um sistema operacional que possuem privilégios elevados para que possam executar rotinas típicas de administração desse sistema.

autorizadas. Como todas as mensagens são retransmitidas para o atacante antes de chegarem ao seu destino, o mesmo tem a possibilidade de adulterá-las ou até mesmo de injetar novas mensagens. Para que o atacante possa realizar esse ataque, ele deve explorar alguma vulnerabilidade presente nos protocolos de comunicação sendo executados nas redes pelas quais as mensagens são transmitidas para alcançar seu destino.

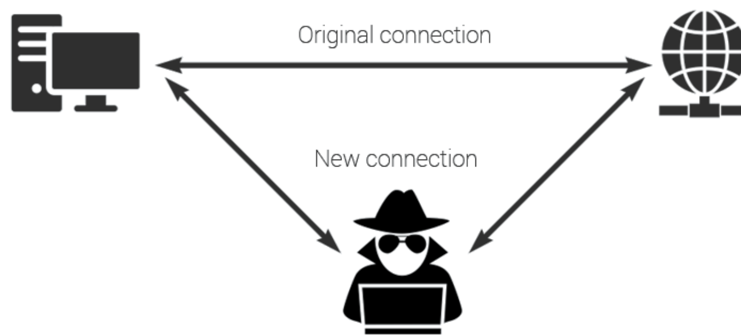

Figura 2 – Ataque *man in the middle*.

Por exemplo, o *arpspoof* é um ataque do tipo *man in the middle* que explora uma vulnerabilidade do protocolo ARP[5]. O atacante envia pacotes ARP informando à vítima que o MAC *address* associado ao endereço IP do roteador da rede é o seu MAC e informa ao roteador da rede que o MAC *address* associado ao IP da vítima também é o seu próprio MAC. Dessa forma, a vítima pensa que o atacante é o roteador e o roteador pensa que o atacante é a vítima. Portanto, quaisquer mensagens trocadas entre a vítima e o roteador serão interceptadas pelo atacante. Caso um usuário acesse uma aplicação *web* e sofra esse tipo de ataque, o atacante será capaz de interceptar todas as mensagens trocadas entre o navegador do usuário e o servidor *web* da aplicação. Os mecanismos internos de segurança implementados pela aplicação não importam, porque o problema de segurança está fora dela. Esse ataque demonstra como uma falha de segurança em um elemento externo pode afetar uma aplicação independente de seus mecanismos de defesa internos. Esse problema surge devido a uma vulnerabilidade presente em um protocolo sendo executado em uma das redes pelas quais as mensagens trocadas entre o cliente e o servidor são transmitidas.

A vulnerabilidade do protocolo ARP descrita anteriormente é efeito colateral de algumas decisões arquiteturais adotadas na concepção da Internet. Durante sua concepção e desenvolvimento, muitos pesquisadores entraram em conflito em relação à implementação de seus principais protocolos. O debate girava em torno da complexidade e da eficiência dos protocolos, que são fatores inversamente proporcionais. Um protocolo de comunicação complexo e robusto pode oferecer mais serviços em troca da redução de sua performance. Os projetistas preferiram focar na eficiência e, portanto, implementaram apenas os ser-

[5] O protocolo ARP (*Address Resolution Protocol*) é um protocolo de resolução de endereços da camada de enlace que mapeia o endereço IP de uma interface de rede ao seu endereço MAC correspondente. É utilizado para viabilizar a comunicação entre os computadores de uma rede local.

viços considerados essenciais para a operação dos protocolos. Estes serviços raramente incluíam aspectos relativos a segurança de comunicações.

Como a Internet era uma rede utilizada por pesquisadores para a transferência de trabalhos acadêmicos, os seus usuários possuiam uma confiança mútua entre si. Dessa forma, o projeto da maioria dos protocolos de comunicação utilizados na Internet não se preocupou com a implementação de mecanismos que provêem segurança à informação transmitida. A forma de reparar isso é implementar novas versões dos protocolos (compatíveis com as anteriores) que incluam medidas defensivas que buscam eliminar falhas de segurança existentes, ou utilizar *softwares* externos que atuam como uma camada adicional de segurança para compensar as limitações dos protocolos, como é o caso do *arpwatch*[6]. O problema desta abordagem é que é difícil tornar seguro algo inerentemente inseguro. Seria mais fácil evoluir protocolos que contemplaram segurança da informação em seu projeto desde sua concepção. Essa deficiência presente em diversos protocolos resulta na necessidade de adicionar elementos de segurança que compensam sua insegurança.

Outra classe conhecida de ameaças de segurança que podem prejudicar redes são os ataques DoS (*Denial of Service*). Este tipo de ataque visa tornar a rede, servidor ou outro elemento da infraestrutura inutilizável por usuários legítimos (KUROSE, 2013). Esses ataques se baseiam em explorar vulnerabilidades ou limitações presentes no sistema alvo de modo a exaurir seus recursos (como memória ou processamento) provocando a interrupção de um ou mais serviços essenciais para seu funcionamento. Uma das abordagens mais utilizadas é a emissão de um volume muito grande de pacotes, possivelmente malformados, visando sobrecarregar o sistema, dessa forma tornando-o indisponível para seus utilizadores.

Por exemplo, a RFC[7] 1122 especifica que um computador conectado à Internet deve responder os pacotes ICMP[8] *echo request* recebidos. Em teoria, um computador deve processar estes pacotes e enviar uma resposta ao cliente. Portanto, um atacante pode sobrecarregar o processador e a largura de banda de um alvo através da emissão de um número excepcional de pacotes.

Outro ataque DoS conhecido é o *SYN flood*, que se concentra em explorar uma característica do protocolo TCP. Como o protocolo TCP é orientado à conexão, dois entes comunicantes devem estabelecer uma conexão entre si antes de trocarem mensagens. Esse processo de estabelecimento de conexão consiste em três etapas. A primeira delas exige que o cliente solicite uma abertura de conexão em seu primeiro contato com o servidor.

[6] O *arpwatch* é um programa que monitora a atividade de uma rede local com o intuito de defender seus usuários de um ataque *arpspoof*
[7] RFC (*Request for Comments*) são documentos técnicos desenvolvidos para especificar padrões que devem ser seguidos na implementação de tecnologias relacionadas a Internet.
[8] O ICMP é um protocolo de rede utilizado para gerar informações de controle, como relatórios de erro, mensagens sobre o estado de um roteador ou indicações de rotas alternativas para um pacote.

Essa solicitação é feita por meio de um pacote *TCP SYN*. Em seguida, o servidor emite um pacote *TCP SYNACK* confirmando o recebimento da requisição pela abertura de uma nova conexão e, por fim, o cliente envia um último pacote *TCP ACK* confirmando o estabelecimento da conexão. O *SYN flood attack* consiste em enviar múltiplos pacotes *TCP SYN* solicitando a abertura de uma conexão, sem completar a terceira etapa (KUROSE, 2013). Em resposta à essa requisição, o servidor irá alocar uma série de recursos (variáveis e *buffers* de memória) para uma conexão que nunca será estabelecida, pois foi aberta pela metade. O atacante repete esse procedimento até consumir todos os recursos do servidor.

É muito difícil um único computador ser capaz de gerar um tráfego de pacotes tão grande a ponto de sobrecarregar um servidor. Por esse motivo, um atacante que aplica um ataque DoS geralmente toma controle de vários sistemas computacionais para executar um ataque distribuído (Fig. 3), denominado DDoS (*Distributed Denial of Service*) (KUROSE, 2013).

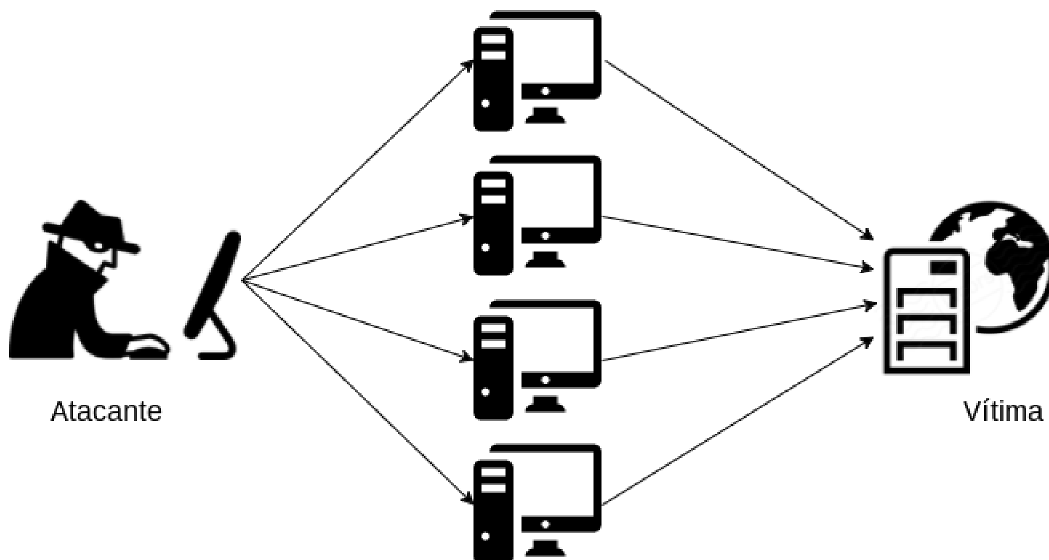

Figura 3 – Ataque DDoS.

2.3.2 Segurança na Rede

Todos os ataques descritos até agora não almejam diretamente um *software* específico, mas os elementos pertencentes ao seu contexto. Contudo, esses ataques indiretos podem comprometer um *software* tanto quanto um ataque direto. Portanto, a segurança de uma aplicação depende da segurança do contexto em que a mesma está inserida.

Como foi demonstrado anteriormente, os dados transmitidos por uma aplicação de rede podem percorrer subredes inseguranças durante seu trajeto. Este fato está fora do controle das aplicações. Portanto, é necessário introduzir protocolos de segurança que implementam mecanismos de defesa que protegem a troca de informações entre duas

entidades, de forma a garantir uma comunicação segura. É possível identificar as seguintes propriedades desejáveis em uma comunicação para que a mesma seja considerada segura:

- Confidencialidade: Apenas o emissor e destinatário devem ser capazes de compreender o conteúdo das mensagens transmitidas (KUROSE, 2013). Como as mensagens trocadas por duas entidades podem passar por redes inseguras, é possível que as mesmas sejam interceptadas por um atacante. Desta forma, é necessário implementar técnicas que tornem as mensagens inintelegíveis para quaisquer entidades que não estejam envolvidas na comunicação. A perda de confidencialidade gera a divulgação não autorizada das informações transmitidas (STALLINGS, 2014).

- Integridade: Deve ser garantido que o conteúdo das mensagens transmitidas não seja alterado, intencionalmente ou não, em trânsito (KUROSE, 2013). Um atacante que intercepte as mensagens associadas à uma comunicação não deve ser capaz de modificá-las sem que o destinatário perceba.

- Autenticidade: Tanto o emissor quanto o destinatário devem ser capazes de confirmar a identidade da outra parte envolvida na comunicação (KUROSE, 2013), ou seja, deve haver uma forma de validar a autoria de uma mensagem.

- Disponibilidade: Os sistemas computacionais e *softwares* utilizados durante a comunicação devem permanecer disponíveis para seus usuários. Por exemplo, um atacante pode tentar depositar programas maliciosos no interior dos computadores de uma rede ou mapear os dispositivos de uma rede e lançar ataques DoS (KUROSE, 2013). Ambos os ataques buscam interromper o funcionamento de um serviço para impedir que usuários legítimos utilizem o sistema.

Existem protocolos de segurança que utilizam técnicas criptográficas para proteger a informação compartilhada e garantir sua confidencialidade, integridade e autenticidade (KUROSE, 2013). No contexto de sistemas *web*, o protocolo HTTP originalmente não provê nenhum tipo de serviço de segurança. A comunicação entre um navegador e uma aplicação *web*, que é governada pelo protocolo HTTP, é suscetível a diversos tipos de ataques (como o *man in the middle*) que comprometem as informações transmitidas, incluindo dados pessoais do usuário, dados bancários, senhas, entre outros tipos de dados sensíveis. Para resolver essa falha de segurança, o protocolo HTTPS (*Hypertext Transfer Protocol Secure*) foi criado. O HTTPS é uma implementação do protocolo HTTP sobre uma camada adicional de segurança que utiliza o protocolo TLS[9]. Essa camada adicional permite que os dados sejam transmitidos por meio de uma conexão criptografada e utiliza

[9] O protocolo TLS (*Transport Layer Security*) é um protocolo de segurança que se situa entre a camada de aplicação e a camada de transporte para implementar funções de segurança que protegem a comunicação entre duas entidades.

um sistema de certificação de chave pública para validar a autoria de uma mensagem. A porta TCP padrão para o protocolo HTTPS é a 443. Estritamente falando, o HTTPS não é um protocolo separado, mas se refere ao uso do HTTP sobre uma camada encriptada de conexão TLS.

Apesar dos mecanismos de defesa introduzidos pelo HTTPS, que de fato implementam três das quatro propriedades desejáveis em uma comunicação segura, ainda existe uma última vulnerabilidade. Esta vulnerabilidade foi exposta por Moxie Marlinspike na conferência *Black Hat* de 2009. O *SSL Strip* é um ataque do tipo *man in the middle* em que um atacante regride o protocolo HTTPS utilizado na comunicação entre uma vítima e um servidor para o protocolo HTTP. Caso o servidor seja configurado para suportar ambas versões, a comunicação entre a vítima e o servidor ocorrerá por meio do HTTP, apesar de ambos suportarem o HTTPS. Dessa forma, o atacante será capaz de ter acesso ao conteúdo da informação sendo compartilhada.

Os navegadores resolveram esse problema introduzindo o HSTS (*HTTP Strict Transport Security*). O HSTS é uma política de segurança que permite a uma aplicação notificar ao navegador que a comunicação entre eles deve ser estritamente segura, ou seja, caso o protocolo HTTPS não seja utilizado, não haverá comunicação. Essa política é uma forma de assegurar que o protocolo HTTP puro não será usado em hipótese alguma. O funcionamento é simples: quando um navegador realiza o primeiro contato com um servidor que adota o HSTS, o mesmo especifica no cabeçalho da resposta que futuras requisições devem seguir o protocolo HTTPS. Isso força o navegador a utilizar o protocolo HTTPS. A maioria dos navegadores incluem antecipadamente uma lista de aplicações que utilizam HSTS.

Como foi citado, atualmente existem meios que viabilizam a segurança das mensagens transferidas entre navegadores e aplicações *web*. Porém, nenhum dos mecanismos apresentados garante a última propriedade desejável em uma comunicação segura: a disponibilidade dos sistemas envolvidos na comunicação. Para implementar essa propriedade, é necessário encontrar uma forma de monitorar o tráfego de pacotes que chegam a uma rede, de modo a identificar pacotes potencialmente maliciosos que podem ter se originado de um ataque *DoS*. É possível incluir dispositivos operacionais à rede, formados por uma combinação de *hardware* e *software*, conhecidos como *firewalls* e *IDSs* (Sistemas de Detecção de Intrusão), que tem o objetivo de inspecionar os pacotes enviados para a rede e filtrá-los conforme suas configurações e políticas de segurança (KUROSE, 2013).

Constatou-se que a segurança de uma aplicação depende dos elementos presentes em seu contexto e que é possível atacá-la de forma indireta através do comprometimento de um destes elementos (como o sistema operacional da aplicação ou a rede de computadores a qual pertence). Foram apresentadas diversas falhas de segurança presentes nestes elementos, ataques que exploram estas vulnerabilidades e mecanismos de defesa que com-

batem esses ataques. Gastou-se um esforço na tentativa de convencer sobre a importância de se levar em consideração a segurança do ecossistema em que um *software* está inserido. Todavia, é lógico que também é preciso levar em consideração a segurança interna do próprio *software*. Não adianta uma aplicação *web* utilizar o protocolo HTTPS para assegurar a comunicação entre o servidor e os seus clientes e não implementar mecanismos de defesa que visam proteger sua lógica interna e os dados que ela manipula. O protocolo TLS protege a confidencialidade e integridade das mensagens em trânsito entre o navegador e o servidor *web*, além de garantir suas autorias. Contudo, ele não impede ataques que alvejam diretamente a aplicação (STUTTARD, 2011).

2.3.3 Segurança Interna

As aplicações precisam interagir de alguma forma com seus usuários e esta interação envolve muitas vezes o recebimento e processamento de dados enviados por eles. Um dos maiores problemas enfrentados pelas aplicações *web* é justamente a possibilidade de os usuários enviarem dados arbitrários (STUTTARD, 2011). Por esse motivo, a aplicação deve assumir que todos os dados recebidos são potencialmente maliciosos e desenvolver mecanismos de segurança que processem e validem esses dados para confirmar que não se trata de um ataque. Além de se preocupar com a entrada dos usuários, muitas aplicações também se preocupam em classificar seus usuários.

Um dos principais requisitos de segurança que diversas aplicações precisa atender é controlar o acesso dos usuários aos seus dados e funcionalidades (STUTTARD, 2011). Um cenário típico inclui diferentes categorias de usuários, como usuários anônimos (não registrados), usuários autenticados e administradores. Cada categoria de usuários possui seu próprio nível de permissões que indica sua capacidade de acessar um determinado conjunto de recursos e funcionalidades da aplicação. A maioria das aplicações *web* gerencia o acesso de usuários através de três mecanismos básicos de segurança: autenticação, gerenciamento de sessão e controle de acesso (STUTTARD, 2011).

A autenticação é um mecanismo que envolve a implementação de uma maneira de identificar um usuário, ou seja, fazer com que a aplicação seja capaz de determinar que um usuário é de fato quem ele afirma ser (STUTTARD, 2011). A maioria dos modelos de autenticação atuais envolve o uso de um nome de usuário ou *email* e uma senha conhecida apenas pelo usuário. Sistemas mais sofisticados cuja segurança é um aspecto crítico do projeto, como aplicações bancárias, podem utilizar dados adicionais para complementar este modelo. Um atacante pode explorar falhas presentes no sistema de autenticação de uma aplicação para acessar recursos e funcionalidades não autorizados.

O gerenciamento de sessão é um mecanismo que busca identificar os usuários associados às suas respectivas requisições. O protocolo HTTP é livre de estados, ou seja, cada par de requisição e resposta é tratado como uma transação independente. Isso quer

dizer que o HTTP não é capaz de estabelecer uma correlação entre uma requisição atual e requisições passadas. A autenticação de um usuário é feita através de uma requisição que contém seu identificador e senha. Após se autenticar, o usuário ganha acesso às páginas da aplicação que não poderia acessar caso não estivesse autenticado. Como o HTTP não é capaz de correlacionar requisições distintas, o protocolo não é suficiente para identificar que as requisições por uma página foram feitas por alguém que já está autenticado ou não.

Para superar essa limitação, as aplicações *web* implementam uma sessão através de *cookies*. Um *cookie* é um arquivo mantido pelo navegador que armazena informações enviadas por um servidor. Um servidor pode enviar no cabeçalho de uma resposta HTTP uma solicitação para que o navegador crie um novo *cookie*. Após a criação de um *cookie*, o navegador o inclui no cabeçalho de futuras requisições destinadas à mesma aplicação. Dessa forma, uma aplicação *web* pode atribuir um *token*[10] aleatório a cada usuário autenticado e solicitar em sua resposta que o navegador crie um *cookie* para armazenar esse identificador. Assim, futuras requisições feitas por esse usuário irão incluir seu *token*, possibilitando que a aplicação seja capaz de confirmar sua identidade.

A vulnerabilidade dessa abordagem é que o *token* de sessão mantido pelo navegador do usuário é uma chave de acesso à sua conta. Um atacante pode se passar por um usuário legítimo caso obtenha seu *cookie*. Um ataque que visa o roubo desse *cookie* de sessão para permitir que um atacante se passe por uma vítima é denominado *session hijacking*. Portanto, é preciso assegurar a comunicação entre o navegador e o servidor, para que os *cookies* estejam protegidos de atacantes que interceptem as mensagens. O HTTP permite que o servidor especifique que um *cookie* deve ser enviado estritamente através de uma comunicação segura. A aplicação deve utilizar um algoritmo que garanta um bom nível de aleatoriedade na geração dos *tokens* que identificam seus usuários, de modo que seja computacionalmente inviável gerar um *token* existente via força bruta[11].

O último mecanismo é o controle lógico de acesso e depende fortemente do funcionamento correto dos dois primeiros. Os dois primeiros mecanismos permitem que a aplicação identifique o usuário que realizou uma determinada requisição. O controle lógico de acesso da aplicação se concentra em avaliar se um usuário possui permissão para acessar a página ou funcionalidade que ele requisitou. O julgamento que examina se um usuário possui permissão para acessar um certo recurso é específico da aplicação. Por exemplo, um usuário comum de uma aplicação *web* de correio eletrônico deve ter acesso apenas à sua própria caixa de *e-mails*. Caso um usuário envie uma requisição à aplicação solicitando acessar a caixa de *e-mails* de outra pessoa, é função da aplicação processar e negar esta requisição. Devido a natureza complexa dos requisitos de controle de acesso,

[10] Um *token* é um valor usado como credencial dentro de um contexto.
[11] Um algoritmo de força bruta testa todas as possíveis soluções de um problema até encontrar a que o satisfaz.

esse mecanismo é uma fonte frequente de vulnerabilidades de segurança que permitem a um invasor obter acesso não autorizado aos recursos de um sistema (STUTTARD, 2011).

Como foi mencionado, o problema central que as aplicações *web* enfrentam é a arbitrariedade dos seus dados de entrada. A principal metodologia adotada pelos atacantes consiste em interagir com o servidor de uma forma inesperada visando interferir no funcionamento normal da aplicação (STUTTARD, 2011). Uma aplicação deve processar e validar seus dados de entrada para garantir que não sejam maliciosos. Esse processamento envolve o uso de *blacklists*[12] contendo um conjunto de *strings*[13] utilizadas em ataques conhecidos, sanitização da entrada que pode ser compreendida como a remoção de um subconjunto de caracteres potencialmente maliciosos, análise semântica para verificar a validade de uma entrada de dados em um determinado contexto, entre outras técnicas (STUTTARD, 2011). Existem vários ataques que exploram o envio de dados maliciosos para comprometer o funcionamento da aplicação, por exemplo, os ataques *SQL injection*, *Cross-Site Scripting* e *Cross-Site Request Forgery*. Todos estes ataques aproveitam falhas de segurança apresentadas pelo sistema *web* durante o processamento dos parâmetros enviados por um usuário em uma requisição.

Um grande número de aplicações fornece um sistema de gerenciamento de conta que permite a um usuário manter e atualizar seus dados pessoais. Quando um usuário deseja utilizar essa funcionalidade para atualizar suas informações pessoais, ele envia seus novos dados atualizados para a aplicação que, por sua vez, realiza o processamento desta requisição para identificar que o usuário solicitou atualizar os seus dados e, portanto, precisa armazenar novos valores no registro correspondente àquele usuário na base de dados do sistema. A atualização da base de dados exige alguma interação entre a aplicação e o sistema de gerenciamento de banco de dados utilizado. Esta interação envolve a emissão de instruções ao sistema de gerenciamento de banco de dados para informar a ação que deve ser feita no banco. Essas instruções devem incluir em algum momento os novos dados que o usuário deseja armazenar. Em um cenário ideal, esses dados serão legítimos, porém, esse nem sempre é o caso. Um usuário mal intencionado poderia enviar instruções direcionadas ao banco no lugar de dados legítimos. Essas instruções seriam embutidas no comando original escrito pelo programador, formando uma nova instrução que pode comprometer a base de dados da aplicação. O ataque *SQL injection* se baseia nesta ideia.

O SQL (*Structured Query Language*) é uma linguagem estruturada destinada à consulta e gerenciamento de bases de dados. As aplicações *web* frequentemente constroem instruções *SQL* para gerenciar o banco de dados do sistema. Essas intruções podem incorporar dados fornecidos pelo usuário (STUTTARD, 2011). O ataque *SQL injection* aproveita o processamento inseguro dos dados de entrada feito por uma aplicação para

[12] Uma *blacklist* é uma lista de itens desaprovados em um dado contexto.
[13] Uma *string* é uma sequência de caracteres.

injetar instruções *SQL* maliciosas com o objetivo de interferir na interação ordinária entre o sistema e a base de dados.

Existem outros tipos de ataques que são direcionados aos usuários da aplicação. É o caso do ataque XSS (*Cross-Site Scripting*). O Javascript é uma linguagem de programação interpretada utilizada pelos navegadores para permitir que os desenvolvedores de uma aplicação *web* programem o comportamento dos elementos que compõem suas páginas, além de oferecer muitas outras funções. O *Cross-Site Scripting* explora a injeção de código Javascript malicioso no interior de páginas *web* legítimas (STUTTARD, 2011). Dessa forma, quando um usuário acessa uma página infectada por esse ataque, seu navegador irá carregar o HTML da página solicitada, incluindo o Javascript embutido pelo atacante. Esse código pode fazer qualquer coisa que esteja ao alcance da linguagem para prejudicar o cliente, por exemplo, redirecionar a vítima para outro *site*, exibir anúncios em benefício do atacante, alterar os elementos da página, obter informações confidenciais, entre outros.

Há duas formas principais de executar este ataque. A primeira delas envolve alguma maneira de interceptar a comunicação entre um cliente e o servidor da aplicação, por exemplo, através de um ataque do tipo *man in the middle*. Caso o atacante seja capaz de interceptar as mensagens transmitidas pelo servidor, ele pode alterar o conteúdo das páginas *web* recebidas pelo usuário e inserir código Javascript malicioso.

A segunda maneira envolve a ausência de processamento e validação da entrada de dados em uma aplicação. Esta versão do ataque surge quando dados enviados pelos usuários são armazenados pela aplicação para serem exibidos posteriormente para outros usuários (STUTTARD, 2011). Por exemplo, em um fórum de notícias, os usuários podem criar tópicos de discussões para permitir que a comunidade da aplicação debata sobre um determinado assunto. Em algum momento, o usuário enviou à aplicação os dados do seu novo tópico, como título e descrição. Estes dados devem ser armazenados pela aplicação em uma base de dados para que posteriormente o sistema possa recuperar estas informações para retorná-las a outro usuário que deseje acessar a página correspondente a este tópico de discussão. Caso a aplicação não filtre a entrada, um usuário poderia injetar código Javascript no lugar de dados legítimos sobre o tópico. Neste cenário, qualquer cliente legítimo que visitasse este tópico provocaria a execução do programa Javascript escrito pelo atacante.

Os ataques *Cross-Site Scripting* podem ser classificados como persistentes (armazenados) e não-persistentes (refletidos). Os ataques persistentes alvejam entradas de dados que serão armazenados no servidor e farão parte da página HTML infectada. Os ataques refletidos ocorrem quando a entrada do usuário é retornada imediatamente pela aplicação como uma mensagem temporária (como o resultado de uma busca ou um código de erro) e, portanto, não afetará a página atacada em futuras requisições.

O Código 2.1 exemplifica um programa malicioso que, caso fosse injetado com sucesso em uma página *web* visualizada por uma vítima, registraria todas as teclas que foram pressionadas por ela enquanto a página estivesse aberta.

Código 2.1 – Javascript que registra as teclas pressionadas por uma vítima e as envia como parâmetros de uma requisição a um domínio controlado por um atacante.

```
<script>
  document.onkeypress = function(event) {
    let img = new Image();
    img.src = 'http://evil-app.com/keylogger?key=' + event.which;
  }
</script>
```

Um outro ataque conhecido é o *Cross-Site Request Forgery* (Fig. 4). Este ataque se assemelha ao *session hijacking* descrito anteriormente, onde um atacante rouba e utiliza o *cookie* de sessão de um usuário para acessar a aplicação se passando pela vítima. O *Cross-Site Request Forgery* não exige que o atacante tenha em posse o *token* de sessão da vítima, mas explora o seu uso para a execução de ações não autorizadas. Este ataque induz um usuário a realizar solicitações que não pretende fazer a uma aplicação em que esteja autenticado (STUTTARD, 2011). O atacante explora a confiança que uma aplicação tem no navegador de um usuário autenticado para induzir usuários legítimos a visitarem *links* maliciosos que realizam solicitações que podem prejudicar a vítima e beneficiar o atacante. Uma das propriedades deste ataque é que os *links* maliciosos podem estar localizados em páginas fora da aplicação almejada.

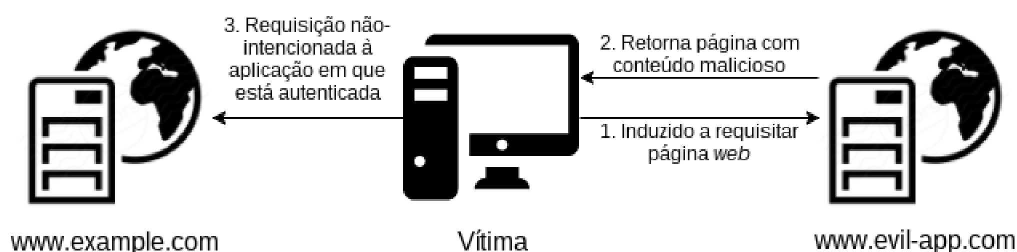

Figura 4 – Ataque CSRF.

Por exemplo, imagine um cenário em que um usuário esteja autenticado na aplicação `www.example.com`. Assume-se que esta aplicação possui um sistema de gerenciamento de conta que pode ser acessado por um usuário através do *link* `www.example.com/account`. Uma das ações oferecidas ao usuário é a de encerrar sua sessão, ou seja, realizar o *logout*. O encerramento da sessão é uma ação privilegiada e está associada à um usuário específico, portanto, exige um *token* de sessão. Caso um atacante induza um usuário a acessar o *link* `www.example.com/account/logout`, o usuário irá realizar uma ação que não pretendia em função das intenções do atacante. Este raciocínio pode se estender para ataques mais

severos, como a realização de compras não intencionadas por uma vítima autenticada em um *site* de comércio eletrônico ou a alteração da senha de um usuário. O poder deste ataque se encontra na dispensabilidade da posse do *cookie* de sessão de um usuário legítimo para realizar ações privilegiadas.

Algumas aplicações empregam o uso de *tokens* de segurança que são gerados a cada requisição e incluídos nas páginas HTML enviadas como resposta aos seus usuários. Dessa forma, as requisições emitidas por um usuário incluem esse *token* para confirmar ao servidor que a solicitação foi feita por um usuário legítimo. Vale notar que este mecanismo de defesa se torna inútil caso um atacante aplique com sucesso um ataque XSS, já que terá acesso ao conteúdo da página, inclusive o *token* de segurança.

Todos os ataques descritos nesta seção visam comprometer de alguma forma um *software*. Existem diversos mecanismos de defesa que podem ser implementados no intuito de proteger uma aplicação. No contexto de aplicações *web*, os principais mecanismos incluem formas de garantir a comunicação segura entre um navegador e o servidor, de identificar a origem de uma requisição e de validar a entrada de dados fornecida por um usuário para mitigar as chances de dados maliciosos prejudicarem o sistema. Entretanto, é difícil antecipar todas as formas possíveis em que um usuário mal-intencionado pode interagir com o aplicativo (STUTTARD, 2011). Portanto, a implementação destes mecanismos se torna um processo de evolução contínua que procura fortalecer cada vez mais o *software* e garantir um certo nível de segurança que transmita aos seus usuários confiança.

2.4 Testes de Penetração de Software

O teste de *software* é uma área de conhecimento da engenharia de *software* que consiste na verificação dinâmica de que um programa apresenta o comportamento esperado pelos desenvolvedores em um conjunto de cenários de teste específicos (BOURQUE, 2014). O processo que envolve a construção de testes de *software* pode ser visto como uma forma de avaliar as funcionalidades do sistema e de identificar possíveis falhas resultantes de erros de implementação. Além disso, os testes de *software* são usados para garantir que uma aplicação mantenha seu comportamento mesmo após a modificação do seu código[14].

Existem várias categorias de teste que buscam classificar os testes em função de seus respectivos objetivos. Cada teste é conduzido com o intuito de atingir um objetivo específico, por exemplo, avaliar a forma como os componentes de um sistema estão integrados ou verificar o funcionamento de um módulo isolado da aplicação. Diversos programadores adotam a prática de automatizar a execução de testes, isto é, escrever programas que objetivam testar um sistema. Esta prática traz diversos benefícios, como a

[14] Um sistema está em constante evolução e a adição de novas funcionalidades vem através da implementação de novo código.

agilidade na execução dos testes (que levariam muito mais tempo se tivessem de ser feitos manualmente) e a documentação através de código do comportamento esperado para uma parte do sistema.

O conceito de teste de *software* pode ser estendido para implementar formas de avaliar a segurança de uma aplicação. Um teste de penetração consiste em um conjunto de métodos e procedimentos que visa testar a segurança de um sistema (BALOCH, 2015). Os testes de penetração auxiliam uma organização a encontrar possíveis vulnerabilidades em seu sistema que podem ser exploradas por um atacante. Os testes de penetração simulam a aplicação de ataques direcionados ao sistema para verificar como será sua reação (BALOCH, 2015). Essas simulações auxiliam os programadores a identificar se o sistema está vulnerável a algum tipo de ataque. Caso esteja, a equipe pode tomar certas medidas para introduzir mecanismos de segurança que buscam defender a aplicação destes ataques.

Uma das razões pela qual as aplicações *web* são tão propensas a ataques é devido a política de segurança da maioria das organizações ser reativa (ANSARI, 2015). Os desenvolvedores de um sistema implementam mecanismos de segurança em resposta a um ataque que explora vulnerabilidades desconhecidas, ou seja, a equipe aguarda algo dar errado para começar a agir e corrigir a falha exposta pelo ataque aplicado. Os testes de penetração fornecem um paradigma alternativo, em que os programadores passam a adotar uma abordagem proativa onde se colocam no lugar do atacante para simular ataques que testam a segurança de um sistema.

A natureza dos testes de penetração ocasiona o estabelecimento de algumas restrições que devem ser acordadas entre a equipe de testes e a organização responsável pelo *software*. A execução dos testes inclui a aplicação de ataques que buscam comprometer o sistema. Nesse sentido, é necessário definir o escopo e os objetivos dos testes, quais partes do sistema serão atacadas, os tipos de ataques permitidos e a metodologia adotada (BALOCH, 2015).

Os testes de penetração podem ser classificados em função da quantidade de informações oferecidas à equipe de testes sobre o *software* a ser testado. As categorias são *black*, *gray* e *white box*. Os testes *black box* ocorrem quando não há praticamente nenhuma informação sobre o sistema em posse da equipe de testes, ou seja, a equipe não tem ciência do sistema operacional executado na máquina da aplicação almejada, do servidor *web* utilizado, da linguagem de programação usada para escrever o sistema, nem do código-fonte do *software* (BALOCH, 2015). Em outras palavras, no teste *black box* a equipe de testes está na mesma posição da maioria dos atacantes. Em contrapartida, os testes *white box* ocorrem quando praticamente todas as informações sobre o sistema são fornecidas à equipe de testes. Por fim, os testes *gray box* são o meio termo entre as categorias *black* e *white* box. Estes testes ocorrem quando algumas informações são fornecidas enquanto

outras são omitidas.

Apesar dos testes de penetração serem recomendados, eles apresentam certas limitações. A qualidade dos testes e os resultados obtidos dependem diretamente da habilidade e conhecimento da equipe de testes. Além disso, as vulnerabilidades encontradas dependem do escopo dos testes e do acesso que a equipe possui ao sistema (quais ataques são permitidos e quais partes podem ser testadas).

Em aplicações *web*, é possível testar qualquer local em que um usuário envia parâmetros para o servidor. Isso inclui os parâmetros da URL solicitada, o cabeçalho da mensagem HTTP enviada, os campos de formulários, entre outros. A aplicação de testes de penetração a esses pontos de entrada pode revelar, por exemplo, falhas de XSS, SQL *injection*, *session hijacking* e *buffer overflow* (ANSARI, 2015). A análise dos resultados desses testes não deve se resumir a observar se houve interrupções no funcionamento da aplicação ou não. É preciso verificar como a aplicação reagiu a certas entradas. Uma das formas de realizar essa verificação é analisar os códigos de *status* das respostas HTTP enviadas pelo servidor. Cada código possui um significado que pode ser usado para inferir vulnerabilidades que podem ser exploradas na aplicação (ANSARI, 2015).

2.5 Grafos

Um grafo é uma estrutura matemática formada por um conjunto de elementos denominados vértices e um conjunto de pares ordenados, denominados arestas, que descrevem relações entre seus vértices. Um grafo pode ser implementado em um programa por diversas estruturas de dados[15] distintas. Cada estrutura apresenta suas vantagens e desvantagens.

Por exemplo, um grafo pode ser representado por uma matriz de adjacências, onde seus vértices são índices inteiros e suas arestas são os valores armazenados no conjunto de pares ordenados $S = \{(i, j) \mid i, j \in \mathbb{N}\}$ da matriz que indicam a existência de uma relação entre os vértices i e j. O Código 2.2 apresenta um programa escrito em linguagem C++ que implementa um grafo através de uma matriz de adjacências, onde os valores *booleanos*[16] armazenados pela matriz descrevem se existe uma conexão entre dois vértices. A Figura 5 fornece uma representação gráfica do grafo deste exemplo.

[15] Uma estrutura de dados consiste em uma coleção de dados organizados e armazenados de uma maneira específica.

[16] Um valor *booleano* é um tipo de dado que pode assumir apenas dois valores possíveis.

Código 2.2 – Matriz de Adjacências.

```
const n = 5;
bool graph[n][n] = {
    {0, 1, 1, 0, 0},
    {0, 0, 1, 1, 0},
    {0, 1, 0, 0, 0},
    {0, 0, 1, 0, 1},
    {0, 0, 0, 1, 0}
};

int main() {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (graph[i][j]) {
                // i is connected with j
            }
}
```

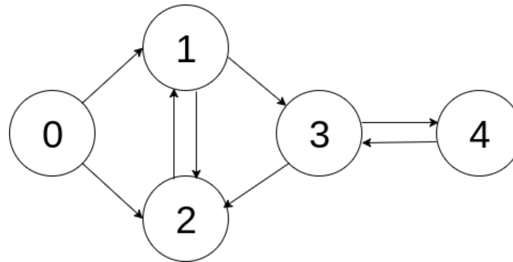

Figura 5 – Representação visual do grafo descrito no Código 2.2.

Em alguns problemas envolvendo grafos, o grafo descrito pelo contexto não exige a utilização de uma estrutura de dados para representá-lo, pois todos os seus vértices e arestas são conhecidos antecipadamente. Os grafos desse tipo são denominados implícitos, pois seus vértices e arestas estão implícitos no contexto do problema.

Este é o caso de um conjunto de páginas *web*, pois as páginas (vértices) e *links* (arestas) entre páginas são conhecidos de antemão. Os *links* de uma página podem referenciar páginas externas à aplicação, ou seja, páginas que pertencem à outra aplicação. Como a própria *web* é formada por todo o conjunto de páginas *web* acessíveis através da Internet, ela pode ser visualizada como um grafo implícito. As páginas que pertencem à uma única aplicação formam um subconjunto deste vértices deste grafo.

A Figura 6 fornece uma representação visual de um grafo que descreve um conjunto de páginas *web*.

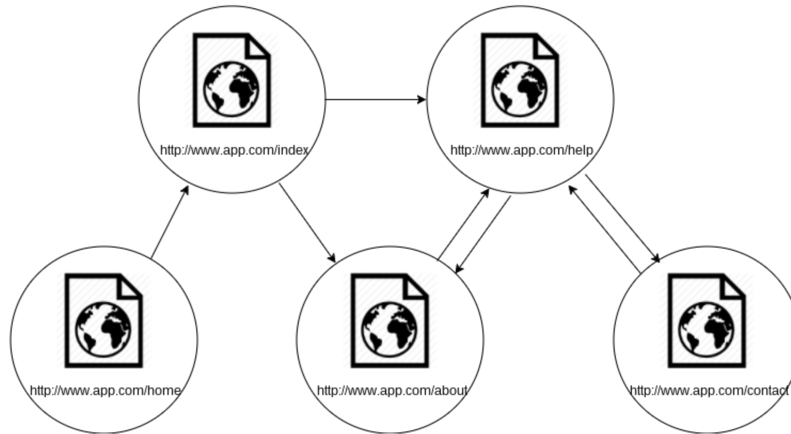

Figura 6 – Grafo que representa um conjunto de páginas *web*.

2.5.1 Algoritmos de Travessia

Um algoritmo de travessia consiste em um algoritmo que implementa um processo de visitação aos vértices de um grafo seguindo um percurso específico. Existem diversos algoritmos de travessia em grafos, por exemplo o BFS (*Breadth-First Search*), também chamado de busca em largura. Cada algoritmo emprega sua própria heurística de visitação de vértices.

O algoritmo BFS se baseia em explorar de forma exaustiva todas as adjacências não visitadas de um vértice de origem, retomando este processo para os novos vértices encontrados seguindo a ordem de descoberta. Este procedimento é repetido até que não haja mais vértices a serem visitados (HALIM et al., 2013). O BFS faz uso de uma fila para garantir a ordem de visitação e um vetor auxiliar que armazena um valor *booleano* para indicar se um vértice já foi visitado ou não.

O Código 2.3 fornece uma implementação do algoritmo BFS aplicado ao exemplo apresentado anteriormente (Código 2.2).

Código 2.3 – Algoritmo BFS.

```
bool visited[3] {false, false, false};

void bfs(int source) {
    std::queue<int> q;
    q.push_back(source);
    while (not q.empty()) {
        int u = q.front(); q.pop();
        for (int v = 0; v < n; ++v) {
            if (graph[u][v] and not visited[v]) {
                visited[v] = true;
                q.push_back(v);
            }
        }
    }
}
```

3 Metodologia

O presente trabalho se concentra no desenvolvimento de um *software*[1] escrito em Ruby[2] que visa automatizar a aplicação de testes de penetração que verificam possíveis vulnerabilidades a ataques XSS em um sistema *web*. O objetivo desse tipo de teste é avaliar se é possível injetar com sucesso um código malicioso em uma aplicação que apresenta falhas no processamento e validação de seus dados de entrada.

O procedimento padrão de um ataque XSS consiste em identificar os pontos de entrada de uma aplicação e injetar código malicioso em cada um destes pontos. Considera-se os campos de entrada dos formulários presentes em uma página HTML como seus pontos de entrada. A identificação destes pontos é feita através da análise sintática do código HTML para localizar seus formulários. Após alvejar os pontos de entrada, é possível verificar se o código injetado é executado quando a página atacada é acessada. Como o projeto descrito neste trabalho busca automatizar a aplicação de ataques XSS a um sistema *web*, é necessário encontrar uma forma de automatizar a localização dos pontos de entrada desse sistema. Após a identificação destes pontos de entrada, basta aplicar os ataques XSS à cada um deles.

Uma aplicação *web* apresenta suas funcionalidades aos seus usuários através de um conjunto de páginas *web*. Cada página *web* é formada por uma série de elementos que são descritos por um código HTML. Toda interação entre um usuário e uma aplicação *web* é feita através de requisições HTTP, onde o usuário solicita um recurso à aplicação e o servidor da aplicação retorna uma página *web* em resposta. Desse modo, as páginas de uma aplicação são acessadas através da realização de requisições ao seu servidor. Uma das principais formas de requisitar uma página da aplicação é inserir sua respectiva URL no campo de endereço do navegador. Todavia, existem maneiras alternativas de realizar a requisição de uma página *web*. Alguns dos elementos HTML de uma página realizam requisições implícitas à aplicação em resposta às ações do usuário.

Por exemplo, uma página *web* pode apresentar *links* para outras páginas através da *tag* HTML `a`. Esta *tag* permite a construção de uma referência à outra página e os usuários da aplicação podem acessar a página referenciada clicando nesta *tag*. Um outro tipo de elemento comum em páginas *web* que fornece uma interação direta entre um usuário e a aplicação é o formulário HTML. Um formulário é um objeto da página formado por um conjunto de campos de entrada que permite que um usuário submeta informações à aplicação. A *tag* HTML que descreve um formulário é a *tag* `form`. As duas *tags* descritas exemplificam objetos de uma página *web* que permitem a um usuário interagir com a

[1] O código-fonte pode ser acessado no *link* https://gitlab.com/pedrodelyra/sekureco.
[2] Ruby é uma linguagem de programação orientada à objetos interpretada que possui tipagem dinâmica.

aplicação.

As interações entre um usuário e a aplicação envolvem a submissão de dados ao servidor e, portanto, são pontos de entrada do sistema *web*. Dessa forma, é possível realizar a análise sintática do código HTML de uma página *web* para identificar os elementos que atuam como pontos de entrada da aplicação, ou seja, permitem que um usuário interaja com o sistema.

A identificação dos pontos de entrada presentes em uma página *web* da aplicação permite a execução de ataques XSS nestes pontos. Contudo, apenas os pontos de entrada desta página serão testados. Como foi dito anteriormente, uma aplicação *web* pode ser formada por inúmeras páginas e, por isso, é necessário pensar em um modo de visitar todas as páginas da aplicação de forma automatizada para que seja possível localizar e testar todos os seus pontos de entrada.

A maioria das páginas *web* possui algum tipo de referência (através de *links*, por exemplo) para outras páginas da aplicação que permitem que um usuário as acesse. Nesse sentido, é possível interpretar um conjunto de páginas *web* como um grafo, onde as páginas são os vértices do grafo e as referências entre as páginas são suas arestas. Seguindo esse raciocínio, a visitação das páginas da aplicação de resume a realizar uma travessia em um grafo, ou seja, descrever um percurso que visite todos os seus vértices. É possível filtrar apenas os *links* que referenciam o domínio da aplicação alvejada, de modo a não visitar páginas externas. Desse modo, a aplicação a ser testada é interpretada como um subconjunto da própria *web* (que também pode ser vista como um grafo) e a varredura se restringe ao *site* alvo.

Portanto, para navegar por todas as páginas do sistema *web* a ser testado, basta executar um algoritmo de travessia sobre o grafo implícito modelado. O código HTML de cada página visitada é processado por um *parser*[3] que extrai seus pontos de entrada (elementos HTML que aceitam parâmetros de entrada do usuário) para serem submetidos à uma bateria de ataques XSS. O algoritmo de travessia escolhido para visitar as páginas da aplicação a ser testada foi o BFS. Este algoritmo é a essência da ferramenta desenvolvida neste trabalho.

A primeira dificuldade nesta abordagem é definir com precisão quando uma página já foi visitada. Como uma página *web* é interpretada como seu código HTML, é possível utilizar uma das estruturas de dados da biblioteca padrão do Ruby, a estrutura *hash*[4], para mapear o código HTML de cada página, que é uma *string*, em um valor *booleano* indicando se a mesma já foi visitada. O Código 3.1 exemplifica o uso da estrutura *hash*

[3] Um *parser* é um programa que realiza o processamento de uma sequência de *strings* seguindo as regras de uma gramática formal.

[4] Uma *hash* é uma estrutura de dados que atua como um *array* associativo que permite o mapeamento entre um par de objetos.

em Ruby. Neste exemplo, uma *string* que armazena um código HTML é fornecida como a chave de uma *hash* para ser mapeada em um valor *booleano*.

Código 3.1 – Estrutura *hash* em Ruby.

```
web_page = "<html><body>...</body></html>"
visited = {}
visited[web_page] = true
```

O problema desse método é que algumas páginas *web* incluem *tokens* de segurança ao longo do seu código. Estes *tokens* são usados para evitar alguns dos ataques descritos no capítulo anterior, como por exemplo o ataque CSRF. Como cada *token* consiste em um código aleatório gerado por requisição para confirmar a legitimidade de uma solicitação, uma página nunca será considerada visitada pela estrutura *hash* (objetos distintos são mapeados em valores distintos), pois duas requisições pela mesma página resultarão em códigos HTML distintos que irão diferir pelos seus *tokens* aleatórios.

O Código HTML 3.2, retirado do formulário de *login* da plataforma Codeforces[5], exemplifica o uso de um *token* de segurança que é gerado a cada requisição pela página de *login*. Nesse sentido, é preciso filtrar o código HTML de cada página antes de marcá-la como visitada, na tentativa de remover elementos aleatórios que não interferem no conteúdo da página.

Código 3.2 – Formulário HTML que inclui um token de segurança.

```
<form method="post" action="/enter" id="enterForm">
    <input type="hidden" name="csrf_token"
           value="f73f67876281076cad4555d2b5b4e864">
    <input id="handle" name="handle" value="">
    <input id="password" name="password" type="password" value="">
    <input class="submit" type="submit" value="Login">
</form>
```

A segunda dificuldade na implementação deste projeto é lidar com os mecanismos de autenticação presentes na maioria dos sistemas *web*. Como foi descrito no capítulo anterior, as aplicações *web* implementam sistemas de autenticação que visam confirmar a identidade de um usuário. Caso o programa desenvolvido neste trabalho não seja capaz de se autenticar na aplicação que deseja testar, ele terá acesso apenas ao conteúdo da aplicação que é público a usuários anônimos. O problema é que este conteúdo público pode ser extremamente limitado, portanto, é necessário implementar uma forma de se autenticar na aplicação que será testada para que seja possível acessar a maior parte de suas funcionalidades e conteúdos.

A autenticação na aplicação pode ser feita após a identificação de algum formulário de *login*. A identificação deste formulário pode se basear em uma lista de nomes comuns

[5] O Codeforces é uma plataforma educacional que pode ser acessada em http://codeforces.com.

presentes nos atributos dos elementos HTML que compõem um formulário de *login*. Após o reconhecimento do formulário de *login*, existem duas abordagens possíveis para concluir a autenticação. A primeira é tentar realizar o *login* via força bruta, ou seja, testar diversas possibilidades de pares de nomes de usuário e senhas até que uma combinação válida seja encontrada. Esta abordagem é ineficiente e poderia levar um tempo aproximadamente infinito caso a aplicação implemente mecanismos básicos de segurança. A segunda abordagem possível, que é a adotada neste projeto, é permitir que os dados de *login* de um usuário existente no sistema *web* a ser testado sejam especificados antes da realização dos testes. Desse modo, o sistema de testes irá utilizar as credenciais fornecidas para iniciar uma sessão na aplicação após a identificação de um formulário de *login*.

A sessão entre um cliente e a aplicação *web* é mantida através de *cookies* HTTP. Portanto, o sistema de testes deve ser capaz de examinar o cabeçalho da resposta HTTP retornada pela aplicação para criar e manter os *cookies* de sessão solicitados. Estes *cookies* devem ser incluídos em futuras requisições por páginas que exigem autenticação. Os mecanismos descritos até então são suficientes para automatizar a navegação em uma aplicação *web*. A navegação automatizada permite que o sistema de testes visite todas as páginas da aplicação alvo.

Com a visitação automatizada às páginas de aplicação, a realização dos ataques XSS exige a identificação dos pontos de entrada de cada página. Esta identificação é feita através da análise sintática do código HTML de cada página. Após a localização destes pontos, basta aplicar um teste XSS em cada um deles.

A versão atual do projeto consiste em quatro módulos principais:

- HTML Page: Este módulo é responsável por abstrair uma página *web* e conta com a biblioteca Nokogiri para realizar a análise sintática do código HTML de uma página. Este componente fornece métodos de acesso aos elementos HTML de uma página e métodos de filtro que extraem elementos indesejados do código HTML.

- HTTP Client: Este módulo se responsabiliza em implementar um cliente HTTP que é utilizado para enviar requisições à aplicação.

- Web Crawler: Este módulo se encarrega de automatizar a navegação pelas páginas de uma aplicação, extrair os pontos de entrada de cada página e aplicar ataques XSS à cada um destes pontos.

- Attacker: Este módulo é responsável por abstrair os ataques que a aplicação é capaz de emitir ao sistema alvo.

A Figura 7 descreve a arquitetura do projeto, seus principais componentes e a forma como interagem entre si.

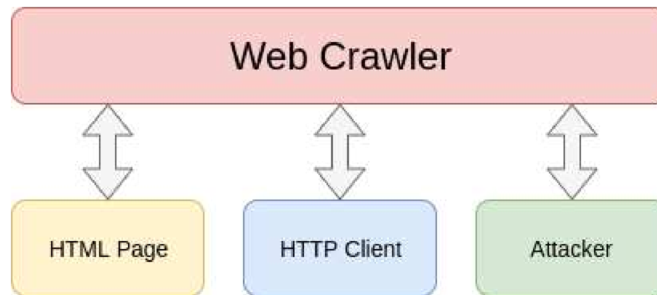

Figura 7 – Arquitetura do projeto.

O Código 3.3 descreve de forma simplificada o algoritmo que automatiza a navegação pelas páginas da aplicação a ser testada.

Código 3.3 – Algoritmo BFS.

```
def crawl
  @queue = [@source_page]
  mark_as_visited @source_page
  @distance = { @source_page => 0 }
  until @queue.empty?
    @current_page = @queue.shift
    next if too_deep? @current_page
    test_forms_of @current_page if @current_page.has_forms?
    @current_page.links.each do |current_link|
      next_link = URI.join(@uri.to_s,
                           URI::encode(parse(current_link)))
      response  = @http_client.get(next_link)
      next_page = HtmlPage.new(response.body)

      unless already_visited? next_page
        mark_as_visited next_page
        update_distance next_page
        @queue << next_page
      end
    end
    check_xss_attack_success
  end
end
```

Em primeiro lugar, o programa inicializa uma fila de vértices inserindo a página pela qual se iniciará a navegação. O algoritmo se repete enquanto houver páginas armazenadas nessa fila. Verifica-se se a página atual está dentro de um raio delimitador em relação à página inicial para evitar que o programa seja executado por uma quantidade de tempo demasiada. Em seguida, os pontos de entrada da página atual são extraídos e

submetidos a ataques XSS. Por fim, o algoritmo examina todos os *links* contidos nesta página para avaliar se existem referências para páginas que ainda não foram visitadas. Caso existam *links* para páginas não visitadas, o algoritmo insere estas páginas na fila para serem processadas posteriormente. Este procedimento se repete até que todas as páginas da aplicação tenham sido visitadas e seus respectivos pontos de entrada tenham sido testados. Para verificar se houve um ataque XSS bem sucedido, as páginas são revisitadas e seus códigos HTML são analisados pelo *parser* para identificar se o código malicioso foi injetado com sucesso na estrutura da página.

Em resumo, a ferramenta de testes funciona da seguinte forma: ela explora as páginas de um *site* em busca de seus pontos de entrada e aplica ataques XSS a cada um deles. Estes ataques incluem um código Javascript com um *token* de identificação gerado aleatoriamente a cada teste. Ao visitar uma página, o teste utiliza este *token* para verificar a presença do código malicioso injetado na página em questão. Caso o código seja encontrado, o ataque é considerado bem sucedido e a página afetada é reportada no *log* do sistema. Como o código é colocado dentro de uma *tag* `script`, é possível utilizar o analisador sintático da ferramenta para verificar se o *script* malicioso pertence à estrutura do código HTML.

4 Resultados e Discussões

A ferramenta foi testada utilizando alguns *sites* destinados a testes de penetração da plataforma educacional *hack.me*[1] e utilizando o Noosfero. O Noosfero é uma plataforma de *software* livre que fornece toda a infra-estrutura necessária para a criação de redes sociais.

4.1 Noosfero

Ao ser usada para testar o Noosfero, o sistema não encontrou nenhuma vulnerabilidade inicialmente. Entretando, ao analisar o código-fonte do Noosfero, verificou-se que o mesmo utiliza um *plugin* denominado *xss_terminate* para se proteger de ataques XSS. Esse *plugin* realiza a sanitização do código HTML das páginas da aplicação.

Para identificar se a ferramenta de testes XSS desenvolvida funciona, esse *plugin* foi desabilitado temporariamente para retirar a imunidade do Noosfero a ataques XSS (Figura 8). Ao desabilitar esse *plugin*, a ferramenta de testes foi capaz de aplicar um ataque com sucesso ao Noosfero.

```
# a base class for all text article types.$
class TextArticle < Article$
$
  def self.short_description$
    _('Text article')$
  end$
$
  def self.description$
    _('Text article to create user content.')$
  end$
$
#   xss_terminate :only => [ :name, :body, :abstract ]
```

Figura 8 – Desabilitação do *plugin xss_terminate*.

Em primeiro lugar, o sistema de testes começa a analisar os pontos de entrada da página inicial do Noosfero. Ao encontrar um formulário de *login* para se autenticar no sistema, a ferramenta utiliza credenciais fornecidas de antemão para iniciar uma sessão no Noosfero. Após isso, um *token* de sessão é mantido para visitar as próximas páginas da aplicação e é enviado no cabeçalho de futuras requisições.

[1] A plataforma *hack.me* possui um acervo de *sites* projetados para serem submetidos a testes de penetração para objetivos educacionais.

A Figura 9 apresenta o *log* inicial do sistema de testes.

```
roots@unknown:~/Workspace/sekureco/lib$ ruby sekureco.rb
Testing http://localhost:3000/


I, [2018-07-09T19:17:21.455874 #22703]  INFO -- : GET http://localhost:3000/

I, [2018-07-09T19:17:22.015145 #22703]  INFO -- : POST http://localhost:3000/account/login

I, [2018-07-09T19:17:22.015221 #22703]  INFO -- : params: {"utf8"=>"✓", "authenticity_token"=
>"RFjM901Wq/yQkTDIQchHNp/2Qlqq8z34rOzJAxdf8DElFlPy516qsAGQjWXPPjHgG9wBFAV5kWtlZ1xO2qoUgg==",
"user[login]"=>"adminuser", "user[password]"=>"admin", "remember_me"=>"1", "commit"=>"Log in"
}

I, [2018-07-09T19:17:22.064663 #22703]  INFO -- : Attempting to apply XSS attack.

I, [2018-07-09T19:17:22.064814 #22703]  INFO -- : POST http://localhost:3000/search/articles
```

Figura 9 – Início do teste.

Em seguida, o sistema de testes realiza uma busca exaustiva examinando as adjacências da página inicial e, posteriormente, as adjacências das adjacências e assim por diante. Em um dado momento, encontrou-se uma página que continha um formulário destinado à criação de artigos. O sistema de testes enviou como entrada deste formulário um ataque XSS (Figura 10). O código deste ataque emite um simples alerta informando que o ataque foi bem sucedido.

```
I, [2018-07-09T21:43:13.815157 #11524]  INFO -- : GET http://localhost:3000/angela-abreu/blog/a
ngela-abreu-doesn-t-use-free-software-no-more?lang=it

I, [2018-07-09T21:43:14.848667 #11524]  INFO -- : Attempting to apply XSS attack.

I, [2018-07-09T21:43:14.848814 #11524]  INFO -- : POST http://localhost:3000/search/articles

I, [2018-07-09T21:43:14.848854 #11524]  INFO -- : params: {"query"=>"<script class='ruzlfdk'>al
ert('Successfully applied XSS attack!');</script>"}

I, [2018-07-09T21:43:14.983211 #11524]  INFO -- : Attempting to apply XSS attack.

I, [2018-07-09T21:43:14.983375 #11524]  INFO -- : POST http://localhost:3000/myprofile/adminuse
r/cms/new_article

I, [2018-07-09T21:43:14.983424 #11524]  INFO -- : params: {"utf8"=>"✓", "authenticity_token"=>"
J/JmSdRWqPVY1Fyn3hVMT8IePl1CuaFC0zAQQo5ipREYVGR0/58tIRJUNh6/ZuuccskjdPDqLeaHc7ZOd2gqrA==", "typ
e"=>"TextArticle", "article[name]"=>"ruzlfdk", "article[body]"=>"<script class='ruzlfdk'>alert(
'Successfully applied XSS attack!');</script>", "commit"=>"Save"}

I, [2018-07-09T21:43:15.326809 #11524]  INFO -- : Attempting to apply XSS attack.
```

Figura 10 – Execução do teste e submissão de um ataque.

A Figura 11 apresenta o *log* do Noosfero onde um artigo com conteúdo malicioso é criado.

```
Started POST "/myprofile/adminuser/cms/new_article" for ::1 at 2018-07-09 21
:43:14 -0300
Processing by CmsController#new_article as */*
  Parameters: {"utf8"=>"✓", "authenticity_token"=>"J/JmSdRWqPVY1Fyn3hVMT8IeP
l1CuaFC0zAQQo5ipREYVGR0/58tIRJUNh6/ZuuccskjdPDqLeaHc7ZOd2gqrA==", "type"=>"T
extArticle", "article"=>{"name"=>"ruzlfdk", "body"=>"<script class='ruzlfdk'
>alert('Successfully applied XSS attack!');</script>"}, "commit"=>"Save", "p
rofile"=>"adminuser"}
```

Figura 11 – *Log* do servidor do Noosfero.

Ao visitar a página do artigo criado, o *log* do sistema de testes informa que uma vulnerabilidade foi encontrada (Figura 12).

```
I, [2018-07-09T21:56:45.122198 #13134]  INFO -- : GET http://localhost:3000/adminuse
r/ruzlfdk?lang=en

I, [2018-07-09T21:56:45.884890 #13134]  INFO -- : Successfully applied XSS attack on
 http://localhost:3000/adminuser/ruzlfdk?lang=en
```

Figura 12 – Vulnerabilidade encontrada.

Após encerrar a visitação das páginas do Noosfero, o *log* da ferramenta informa se houve um ataque bem sucedido. A Figura 13 exibe este *log*.

```
================================================================================
Vulnerabilities found
================================================================================

I, [2018-07-09T22:11:58.852485 #13134]  INFO -- : Test duration: 23m30s
```

Figura 13 – Resultado do teste.

Para confirmar o sucesso do ataque, a página foi visitada utilizando um navegador e verificou-se que o alerta foi emitido (Figura 14) e, portanto, o código Javascript malicioso foi executado de forma bem sucedida.

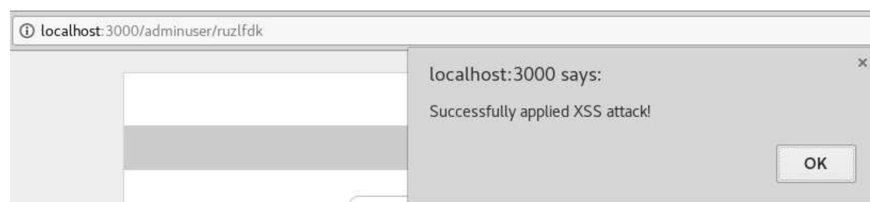

Figura 14 – Código malicioso executado pelo navegador.

A Figura 15 apresenta o código Javascript embutido no HTML da página utilizando o inspetor do navegador.

```
▼<div class="article-body article-body-text-article">
   <script class="ruzlfdk">alert('Successfully applied XSS attack!');</script>
   <br style="clear:both">
 </div>
 <!-- end class="article-body" -->
```

Figura 15 – Código malicioso embutido no HTML da página infectada.

4.2 Hack.me

A ferramenta encontrou vulnerabilidades nos *sites* educativos da plataforma *hack.me*. Abaixo, segue a descrição da execução dos testes XSS em um dos alvos da plataforma *hack.me*.

A Figura 16 descreve o início do teste. A ferramenta parte de uma página inicial e começa a explorar as páginas adjacentes seguindo o algoritmo BFS.

```
roots@unknown:~/Workspace/sekureco/lib$ ruby sekureco.rb
Testing http://s91540-103322-g8h.sipontum.hack.me

[2018-07-14 18:27:35] INFO -- GET http://s91540-103322-g8h.sipontum.hack.me

[2018-07-14 18:27:36] INFO -- GET http://s91540-103322-g8h.sipontum.hack.me/index.html
```

Figura 16 – Início do teste.

A Figura 17 apresenta o momento que a ferramenta tenta aplicar um ataque XSS a um ponto de entrada identificado.

```
[2018-07-14 18:27:54] INFO -- Attempting to apply XSS attack.

[2018-07-14 18:27:54] INFO -- POST http://s91540-103322-g8h.sipontum.hack.me/reflected-XSS/search.php

[2018-07-14 18:27:54] INFO -- params: {"search"=>"<script class='ruzlfdk'>alert('Successfully applied XSS attack!');</script>"}

[2018-07-14 18:27:54] INFO -- Attempting to apply XSS attack.

[2018-07-14 18:27:54] INFO -- POST http://s91540-103322-g8h.sipontum.hack.me/persistent-XSS/persistent-index.php

[2018-07-14 18:27:54] INFO -- params: {"comment"=>"<script class='ruzlfdk'>alert('Successfully applied XSS attack!');</script>"}
```

Figura 17 – Tentativa de ataque XSS.

A Figura 18 descreve o ponto em que o código Javascript injetado é identificado na estrutura do HTML de uma página infectada e, portanto, acusa uma vulnerabilidade.

```
[2018-07-14 18:27:55] INFO -- Successfully applied XSS attack on http://s91540-103322-g
8h.sipontum.hack.me/persistent-XSS/persistent-index.php

[2018-07-14 18:27:55] INFO -- POST http://s91540-103322-g8h.sipontum.hack.me/persistent
-XSS/persistent-index.php

[2018-07-14 18:27:55] INFO -- params: {"clear"=>"Clear Table"}

[2018-07-14 18:27:55] INFO -- Successfully applied XSS attack on http://s91540-103322-g
8h.sipontum.hack.me

[2018-07-14 18:27:55] INFO -- Successfully applied XSS attack
```

Figura 18 – Identificação de um ataque bem sucedido.

A Figura 19 exibe os *logs* finais do teste retornando o resultado e o tempo de execução.

```
================================================================================
Vulnerabilities found
================================================================================

[2018-07-14 18:27:59] INFO -- Test duration: 00m23s
```

Figura 19 – Resultado do teste XSS.

4.3 Considerações

A ferramenta desenvolvida implementa uma varredura básica em um *site* alvo e aplica ataques XSS simples aos seus pontos de entrada. Porém, ela apresenta limitações. Existem várias interações entre um usuário e uma aplicação *web* que demandam um *environment* Javascript (como o do navegador) que não é implementado pela ferramenta. Por exemplo, interações que envolvem eventos específicos com os elementos da página, como um clique, o ato de passar o *mouse* por cima de um elemento, o ato de focar um componente da página, entre outras. Essas interações podem enviar requisições implícitas ao servidor e o estado atual da ferramenta não contempla esse tipo de interação entre o cliente e o servidor.

De todo modo, a ferramenta pode ser evoluída para incluir outros tipos de testes de penetração que envolvem entradas maliciosas, além de ser possível implementar um módulo especial para realizar testes de carga em um sistema *web*. Também é possível fazer a análise de várias propriedades das páginas visitadas, como por exemplo o nível de aleatoriedade dos *tokens* randômicos presentes na página.

5 Conclusão

O crescimento da *web* proporciona o desenvolvimento de cada vez mais aplicações que implementam todo tipo de função, inclusive funcionalidades que envolvem dados sensíveis do usuário. A natureza destas aplicações exige um cuidado na implementação de mecanismos de segurança que visam protegê-las de ataques mal-intencionados. A aplicação contínua de testes de penetração parece ser uma boa alternativa para avaliar o nível de segurança de um sistema, além de contribuir com a identificação de possíveis vulnerabilidades que podem expor a aplicação.

Ao longo deste trabalho, apresentaram-se os principais pontos relacionados à segurança de aplicações *web* atualmente. Abordou-se uma série de ataques que visam comprometer um sistema e formas de defendê-lo destes ataques. Discutiu-se, também, uma abordagem proativa no desenvolvimento de aplicações seguras, a qual exige um estudo constante sobre segurança de *software*.

Foi desenvolvida uma ferramenta para testar ataques XSS do tipo persistente que navega automaticamente pelas páginas de uma aplicação *web*, que localiza seus pontos de entrada e os submete a ataques XSS. As páginas atacadas são examinadas para verificar o sucesso dos ataques. Testou-se a ferramenta em *sites* educacionais da plataforma *hack.me* e no *software* livre Noosfero, sendo que houve ataques bem sucedidos nos alvos do *hack.me* e no Noosfero apenas foram obtidos ataques com sucesso após a desabilitação de um *plugin* que realiza a sanitização do código HTML de suas páginas. Esse resultado comprova a importância e eficiência da sanitização de entrada em aplicações *web*.

Referências

ANSARI, J. A. *Web Penetration Testing with Kali Linux*. [S.l.: s.n.], 2015. Citado 3 vezes nas páginas 19, 39 e 40.

BALOCH, R. *Ethical Hacking and Penetration Testing Guide*. [S.l.: s.n.], 2015. Citado na página 39.

BLAKLEY, B.; MCDERMOTT, E.; GEER, D. Information security is information risk management. In: ACM. *Proceedings of the 2001 workshop on New security paradigms*. [S.l.], 2001. p. 97–104. Citado na página 26.

BOURQUE. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. [S.l.]: IEEE Computer Society Press, 2014. Citado na página 38.

HALIM, S. et al. *Competitive Programming 3*. [S.l.]: Lulu Independent Publish, 2013. Citado na página 42.

KUROSE, J. *Computer Networking A Top Down Approach*. Upper Saddle River, New Jersey: [s.n.], 2013. 889 p. ISBN 978-0-13-285620-1. Citado 9 vezes nas páginas 21, 22, 23, 24, 25, 29, 30, 31 e 32.

MCGRAW, G. *Software Security Building Security In*. [S.l.: s.n.], 2001. 396 p. Citado 2 vezes nas páginas 26 e 27.

STALLINGS, W. *Cryptography and Network Security: Principles and Practice*. Upper Saddle River, New Jersey: [s.n.], 2014. 758 p. ISBN 978-0-13-335469-0. Citado na página 31.

STUTTARD, D. *The Web Application Hacker's Handbook*. Rosewood Drive, Danvers: [s.n.], 2011. 914 p. ISBN 978-1-118-02647-2. Citado 7 vezes nas páginas 25, 27, 33, 35, 36, 37 e 38.

TANENBAUM, A. *Redes de Computadores*. Upper Saddle River, New Jersey: [s.n.], 2014. 563 p. ISBN 978-85-7605-924-0. Citado na página 21.

WHEELER, D. *Secure Programming*. [S.l.: s.n.], 2015. 194 p. Citado na página 27.