UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ANDREY BLAZEJUK

Prevenção de Vulnerabilidades em Aplicações Web Utilizando o Framework Laravel

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Raul Fernando Weber

Porto Alegre
2017

AGRADECIMENTOS

Agradeço a Deus pela Sua graça imensurável e por ter me capacitado ao longo de toda a minha vida para que eu pudesse chegar até onde estou hoje. Trabalhar com computação nem sempre foi o meu objetivo e os últimos 5 anos não foram fáceis, mas acredito que essa sempre foi a Sua vontade e saber disso me ajudou a seguir em frente nos dias difíceis.

Sou grato ao meu pai, Daniel Blazejuk, e à minha mãe, Nair Neumann Blazejuk, que sempre me apoiaram e me incentivaram para que eu me dedicasse a estudar o assunto que fosse do meu interesse. Agradeço por nunca terem me cobrado melhores resultados ou terem me pressionado para que eu me formasse antes. As coisas mais importantes que eu aprendi não estavam nos livros ou nas aulas, mas foram vocês que me ensinaram.

Às demais pessoas da minha família e aos meus amigos, obrigado por sempre desejarem o melhor para mim, desde quando eu estava estudando para o vestibular, e por terem se alegrado comigo após a notícia da minha aprovação. Fico feliz em ter vocês por perto na minha formatura.

Ao meu orientador Raul Fernando Weber, por ser um ótimo professor e ter contribuído para que eu me interessasse pela área de segurança em sistemas computacionais. Obrigado por ter aceitado o convite de ser meu orientador e por me ajudar a focar no assunto abordado neste trabalho.

Agradeço ao Oscar de Souza Dias por ter me apresentado o *framework* utilizado neste trabalho e por ter me dado a oportunidade de trabalhar com desenvolvimento Web. Aos meus colegas de estágio, orientadores e bolsistas de iniciação científica que participaram dos projetos em que trabalhei durante a graduação, vocês também contribuíram significativamente para a minha formação.

Aos meus demais professores, que mostraram domínio inegável sobre o conteúdo abordado em cada disciplina que eu cursei. Vocês facilitaram enormemente o processo de aprendizagem para mim e não é por acaso que são referências nas mais diversas áreas da computação.

RESUMO

A prevenção de vulnerabilidades de segurança é um tópico de estudo em constante evolução. Diariamente, novas vulnerabilidades são exploradas nas mais diversas plataformas e ferramentas. Da mesma forma, os pesquisadores e desenvolvedores frequentemente propõem novos mecanismos para se proteger contra as ameaças que surgem. Com a popularização da internet, aplicações Web se tornaram um dos principais alvos de ataques de usuários mal-intencionados. Diante do desafio de evitar que estes sistemas sejam invadidos, este trabalho se propõe a estudar medidas de prevenção que podem adotadas pelos desenvolvedores para impedir a exploração das vulnerabilidades críticas já conhecidas por atingirem estas aplicações. Este trabalho busca chamar atenção dos desenvolvedores para estas vulnerabilidades, mostrando como os usuários de uma aplicação Web podem ser prejudicados se um ataque for bem-sucedido. Para isso, uma aplicação Web de fórum online foi desenvolvida utilizando o *framework* Laravel, que é popular entre desenvolvedores de sistemas Web, como prova de conceito. Através dos experimentos realizados no sistema, é possível ilustrar como estes ataques são realizados por usuários maliciosos e como podem ser evitados pelos projetistas do software se os recursos oferecidos pelo Laravel forem utilizados corretamente.

Palavras-chave: Segurança. Aplicações Web. Prevenção de vulnerabilidades.

Vulnerability Prevention in Web Applications Using the Laravel Framework

ABSTRACT

Security vulnerabilities prevention is a topic of study in constant evolution. Daily, new vulnerabilities are exploited in many platforms and tools. At the same pace, researchers and developers frequently propose new mechanisms to protect themselves against the incoming threats. With the popularization of the internet, Web applications have become one of the main targets for attacks of malicious users. Facing the challenge of avoiding the invasion of these systems, this work propose a study of the prevention measures that may be adopted by developers to stop the exploiting of critical vulnerabilities already known to affect these applications. This work aims to draw attention of developers to these vulnerabilities, showing how Web applications users could be affected if an attack is successful. To achieve this, an Web application of an online forum was developed using the Laravel framework, which is popular among Web systems developers, as a proof of concept. Through the experiments done in the system, it is possible to illustrate how these attacks are made by malicious users and how they can be avoided by software designers if the resources provided by Laravel are properly used.

Keywords: Security. Web applications. Vulnerabilities prevention.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ASLR	Address Space Layout Randomization
CLI	Command-Line Interface
CSRF	Cross-Site Request Forgery
DAST	Dynamic Application Security Testing
DEP	Data Execution Prevention
DNS	Domain Name System
DOM	Document Object Model
FTP	File Transfer Protocol
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDS	Intrusion Detection System
IPC	Inter-Process Communication
MVC	Model-View-Controller
ORM	Object-Relational Mapping
OWASP	Open Web Application Security Project
PDO	PHP Data Objects
SGBD	Sistema de Gerenciamento de Banco de Dados
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WAF	Web Application Firewall
WYSIWYG	What You See Is What You Get
XSS	Cross-Site Scripting

SUMÁRIO

1 INTRODUÇÃO

Sistemas computacionais são desenvolvidos pelas empresas com o objetivo de atender às necessidades de seus clientes da melhor maneira possível dentro dos prazos acordados por ambos. A maior parte do tempo de desenvolvimento é investida na implementação dos requisitos funcionais do sistema, portanto preocupações com aspectos de segurança naturalmente recebem menor prioridade e menos atenção dos desenvolvedores.

Algumas práticas devem ser adotadas pelos programadores para que vulnerabilidades sejam prevenidas, como: revisar o código fonte a procura de falhas de segurança, executar testes de intrusão e utilizar analisadores de vulnerabilidades (VIEIRA, 2009). O custo de realizar estes procedimentos e corrigir uma vulnerabilidade durante a fase de desenvolvimento é muito menor do que o dano que pode ser causado se um atacante invadir o sistema e acessar os dados sensíveis dos usuários.

Todos os sistemas computacionais devem ser projetados levando em consideração aspectos de segurança, entretanto, existem algumas aplicações que demandam um cuidado maior sobre estes aspectos do que outras. Por exemplo, uma aplicação Web que permita que seus clientes realizem transações financeiras deve ser desenvolvida e testada com mais enfoque na segurança de seus usuários do que um editor de texto comum, que sequer exige acesso à internet para cumprir seu propósito.

1.1 Aplicações Web

Devido à sua natureza de alta disponibilidade, as aplicações Web são um alvo particularmente interessante para usuários maliciosos. A arquitetura deste tipo de sistema demanda que exista um servidor Web, responsável por disponibilizar as páginas do website, que são transferidas pela rede utilizando o protocolo Hypertext Transfer Protocol (HTTP) para serem renderizadas pelo navegador do usuário. Cabe a lógica da aplicação executada neste servidor realizar verificações de segurança para garantir que os dados serão acessados e modificados apenas por usuários que possuem as permissões necessárias para realizar tais ações.

Qualquer erro na configuração do servidor ou na validação das entradas do usuário pode permitir a execução de código malicioso no servidor ou tornar os dados sensíveis dos usuários públicos. Não existe uma solução completa que garanta que uma aplicação Web está livre de vulnerabilidades e, ainda que existisse, esta solução não seria suficiente para garantir

a proteção dos usuários devido à variedade e à efetividade de ataques de engenharia social, como é possível observar em (KROMBHOLZ, 2015) e (IRANI, 2011).

Este trabalho tem como objetivo estudar e ressaltar a importância de medidas de prevenção que podem e devem ser utilizadas contra as principais vulnerabilidades em aplicações Web. Se os desenvolvedores estiverem cientes dos riscos existentes e empregarem as medidas corretas para evitá-los desde o início do desenvolvimento do sistema, muitos dos possíveis ataques direcionados a aplicação serão frustrados.

1.2 Organização do Trabalho

A organização dos próximos capítulos é a seguinte. No Capítulo 2, os principais conceitos e tecnologias relacionados com a segurança de sistemas Web serão aprofundados. O terceiro capítulo apresenta 5 das principais vulnerabilidades críticas para estes sistemas: SQL Injection, Quebra de Autenticação e Gerenciamento de Sessão, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF) e Referência Insegura e Direta a Objetos. Neste capítulo, exemplos de erros que levam a estas vulnerabilidades são citados, justificando a importância de evitá-las.

Em seguida, serão descritas as técnicas e medidas de prevenção conhecidas até o momento para combater as vulnerabilidades apresentadas anteriormente. O Capítulo 5 ilustra a exploração e a prevenção de 2 das vulnerabilidades apresentadas ao longo do trabalho (SQL Injection e XSS) através da implementação de uma prova de conceito. A prova de conceito é uma aplicação Web que implementa um fórum de discussão online, desenvolvida utilizando o *framework* Laravel. A Seção 5.1 apresenta o funcionamento geral da aplicação. A Seção 5.2 ilustra como um atacante pode explorar a vulnerabilidade de SQL Injection no processo de login e a vulnerabilidade de XSS na funcionalidade de pesquisa em tópicos e postagens. Estas vulnerabilidades foram introduzidas no sistema pelo uso incorreto dos recursos oferecidos pelo Laravel. Na Seção 5.3, estas 2 vulnerabilidades são corrigidas aplicando as medidas de prevenção citadas no Capítulo 4 e utilizando o *framework* adequadamente.

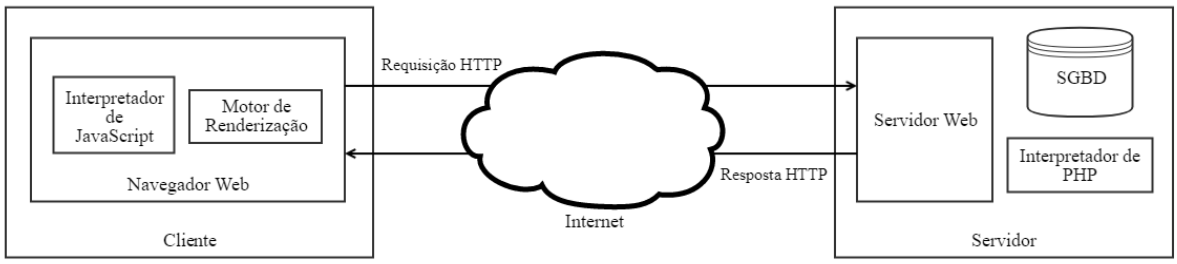
2 SEGURANÇA EM SISTEMAS WEB

Na internet, corporações e agências do governo mantém uma grande quantidade de dados, que devem ser protegidos de criminosos em um ambiente heterogêneo e distribuído, onde é fácil para um usuário permanecer anônimo e *malwares* são distribuídos sem grandes dificuldades (JOSHI, 2001). Ainda que ataques internos em intranets corporativas sejam uma preocupação, a identificação do atacante, que neste caso é um funcionário da empresa que detém o sistema, é mais simples do que na internet devido a quantidade reduzida de usuários suspeitos e ao acesso restrito ao sistema.

Com a popularização de aplicações de *e-commerce*, transações financeiras se tornaram cada vez mais frequentes. Ao realizar uma compra online, o usuário fornece informações sensíveis como seu número de cartão de crédito e seu endereço residencial. Estas informações devem ser protegidas desde quando o usuário preenche o formulário com seus dados utilizando seu computador pessoal até o armazenamento dos mesmos no banco de dados mantido pela loja.

Em geral, sistemas Web seguem a arquitetura cliente-servidor (Figura 2.1). A estrutura deste sistema é composta por um cliente com um navegador Web instalado, que envia requisições HTTP através da internet para um servidor Web, que, na maioria dos casos, é conhecido previamente fazendo uso do protocolo Domain Name System (DNS). Após receber uma requisição, o servidor processa os dados recebidos utilizando um interpretador de uma linguagem de script (como PHP, por exemplo), que pode realizar consultas ao banco de dados, e envia uma resposta ao cliente. O cliente, por sua vez, é responsável por exibir o resultado processado e interpretar o código JavaScript da página no navegador do usuário final.

Figura 2.1 – Arquitetura típica de sistemas Web



Fonte: Próprio autor.

Uma falha de segurança em qualquer um dos componentes desta arquitetura é suficiente para colocar todo o sistema em uma situação de risco. Para que o sistema seja protegido, é necessário analisar individualmente se cada um dos agentes deste modelo atende aos princípios de segurança fundamentais para desempenhar seu papel sem que informações sigilosas sejam expostas.

2.1 Princípios de Segurança

Com o objetivo de tornar sistemas Web mais seguros, é necessário definir o conceito de segurança computacional, que abrange um conjunto maior de aplicações e arquiteturas, mas também se aplica aos sistemas estudados neste trabalho. A segurança computacional consiste na proteção contra danificação ou roubo de hardware e de informações, incluindo o esforço em evitar a interrupção de serviços prestados (GASSER, 1988).

Três princípios fundamentais de segurança devem ser aprofundados para complementar a definição acima: confidencialidade, integridade e disponibilidade. Um sistema que atenda a estes princípios, nas proporções adequadas ao seu contexto, pode ser considerado protegido. O estudo de segurança envolve ainda a identificação de ameaças, mecanismos de detecção e prevenção de ataques e análise de riscos. Apesar dos esforços existentes na conscientização dos usuários e desenvolvedores dos sistemas, a parte mais vulnerável de qualquer sistema são as pessoas que interagem com ele (BISHOP, 2005). Por isso, as políticas e os procedimentos de segurança devem levar em consideração as pessoas que terão acesso ao sistema.

2.1.1 Confidencialidade

A confidencialidade se baseia em impedir que informações privadas sejam divulgadas a pessoas que não estão autorizadas a recebê-las. As motivações para manter dados ocultos de outras pessoas podem ser diversas, desde uso militar até estratégias de empresas para se manter à frente dos concorrentes no mercado em que atuam. Seja qual for o motivo para se empregar a confidencialidade, seu uso demonstra que existem informações sigilosas sendo protegidas.

A criptografia é o principal mecanismo utilizado para a obtenção de confidencialidade. Através de uma chave criptográfica, utilizada por duas partes querendo se comunicar, as

mensagens transmitidas por um emissor são compreendidas apenas pelo seu receptor, de modo que um terceiro agente que intercepte a comunicação não consiga decifrar o conteúdo das mensagens sem o conhecimento desta chave.

2.1.2 Integridade

O princípio da integridade trata da correção e da confiança a respeito dos dados recebidos, além de impedir que alterações impróprias ou não autorizadas sejam feitas sobre estes dados. Este conceito não se limita apenas as informações obtidas, mas também inclui a origem destas informações. A integridade da fonte dos dados é conhecida como autenticação.

Os mecanismos de integridade podem ser de prevenção ou de detecção. A prevenção deve assegurar que alterações não sejam efetuadas por um usuário sem autorização e que usuários autorizados não realizem mudanças indevidas. A detecção permite a identificação de que os dados não são confiáveis. Uma técnica amplamente utilizada para detecção de dados corrompidos é o uso de *checksums*.

2.1.3 Disponibilidade

A capacidade de utilizar a informação ou o recurso desejado define o conceito de disponibilidade. Se um sistema estiver em um ambiente diferente daquele para o qual ele foi projetado, falhas ocorrerão. Em algumas situações, é possível que falhas sejam provocadas propositalmente com o objetivo impedir o acesso a um serviço ou às informações.

Ataques de negação de serviço podem reduzir o desempenho de um sistema temporariamente, tornar o sistema indisponível até que seja reinicializado manualmente ou causar a perda de dados sem que seja possível recuperá-los. A identificação deste tipo de ataque não é trivial, já que as causas para estes comportamentos podem ser naturais, devido a características do ambiente de execução, ou intencionais, quando estimulados através da manipulação de recursos.

2.2 Navegadores Web

Na Web, recursos como páginas Web ou imagens são identificados através de Uniform Resource Identifiers (URIs). Os navegadores são aplicações desenvolvidas para que

estes recursos sejam recuperados utilizando a rede e exibidos ao usuário, que é capaz de acessar novos recursos através de *hypertext links*.

Navegadores Web, como qualquer outro software, não estão livres de *bugs*. Estas aplicações são utilizadas para acessar a internet, que é naturalmente um ambiente hostil, e lidam com dados sensíveis que devem ser armazenados de forma segura, como senhas e dados bancários dos usuários. Portanto, estes softwares devem ser projetados com o objetivo evitar ao máximo que erros de programação permitam a exploração de uma vulnerabilidade e que, caso a exploração ocorra, o dano ao sistema do usuário seja o menor possível.

Um mecanismo de segurança empregado para evitar que um atacante bem-sucedido instale malwares ou leia arquivos do disco rígido do usuário, por exemplo, é o uso de *sandboxes*. Este mecanismo consiste na virtualização de um ambiente de execução isolado, onde os processos possuem acesso limitado ao sistema de arquivos e as demais funcionalidades do sistema operacional.

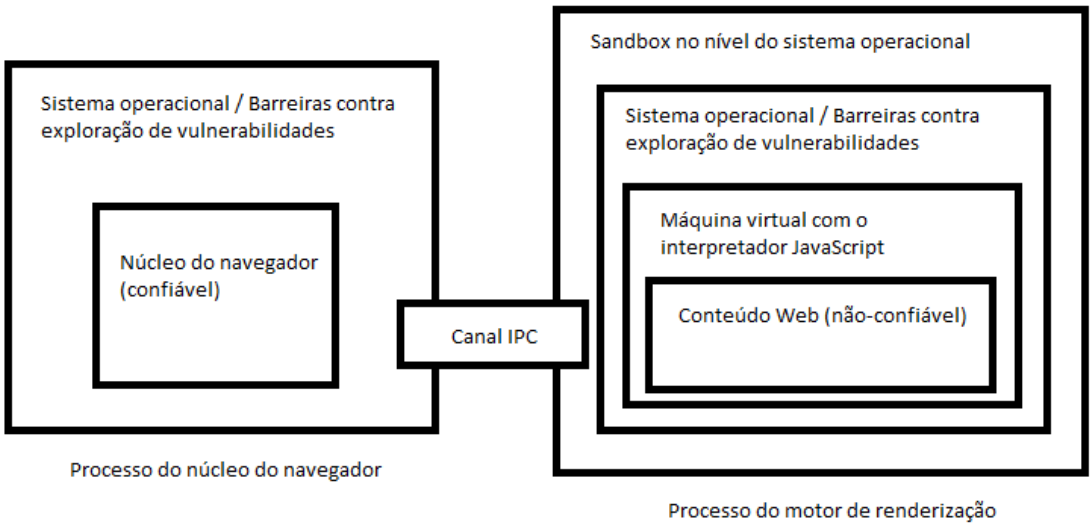
A arquitetura de segurança utilizada no desenvolvimento do Google Chrome apresentada em (REIS, 2009) é composta por dois processos principais escalonados pelo sistema operacional: o núcleo do navegador (com alto nível de privilégios) e o motor de renderização (com baixo nível de privilégios).

O núcleo do navegador é responsável por desenhar a interface do usuário, armazenar bases de dados de *cookies* e do histórico de navegação e por fornecer acesso à internet. O motor de renderização realiza o *parsing* do código Hypertext Markup Language (HTML), executa o código JavaScript da página, decodifica imagens e realiza as demais tarefas necessárias para que as páginas Web sejam exibidas.

Este processo menos privilegiado é isolado dos demais através de uma *sandbox* em nível de sistema operacional. Adicionalmente, existe uma máquina virtual interna a este processo utilizada para que o código JavaScript de cada site seja interpretado individualmente. A comunicação entre os dois processos é realizada pela troca de mensagens através de um canal Inter-Process Communication (IPC). A arquitetura de segurança do Google Chrome é apresentada na Figura 2.2.

Para tornar a exploração de vulnerabilidades mais difícil, barreiras como Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR) e GS são utilizadas pelo Chrome. DEP marca páginas de memória como não-executáveis, ASLR randomiza os endereços de memória alocados para as áreas que compõem o processo e GS é uma opção de compilação que auxilia na detecção de alterações do endereço de retorno de funções na pilha.

Figura 2.2 – Arquitetura de segurança do Google Chrome



Fonte: Reis (2009, p. 46).

Plug-ins adicionados ao navegador (que ocupem a tela inteira ou acessem periféricos como a webcam ou o microfone, por exemplo) são executados fora da *sandbox* por necessitarem da interface direta com o sistema operacional para funcionarem corretamente. Portanto, a segurança dos *plug-ins* fica sob a responsabilidade de seus desenvolvedores e não dos desenvolvedores do navegador.

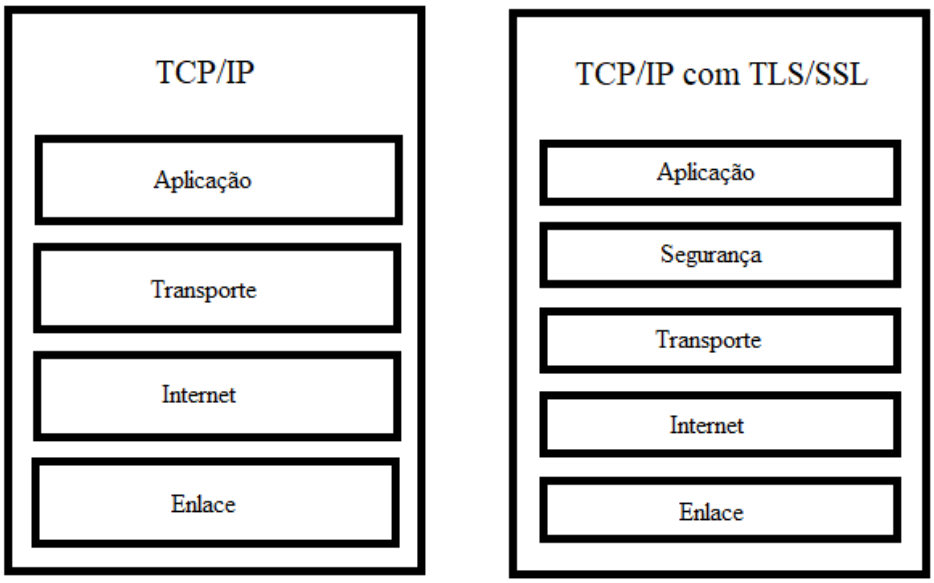
2.3 Comunicação

O Transmission Control Protocol (TCP) é um protocolo da camada de transporte do modelo TCP/IP, que é orientado a conexões e fornece o serviço de transporte confiável entre dois processos executando em *hosts* distintos. Técnicas como detecção de erros, retransmissão de mensagens, temporizadores e números de sequência são usadas para alcançar a confiabilidade.

A versão do protocolo TCP que implementa serviços de segurança é conhecida como Secure Sockets Layer (SSL), que a partir de sua terceira versão passou a se chamar Transport Layer Security (TLS). SSL faz parte da camada de aplicação, mas sua utilização pelos desenvolvedores é feita pela inclusão de bibliotecas ou classes que oferecem uma Application Programming Interface (API) com *sockets* semelhante a do TCP. O uso de SSL adiciona ao TCP confidencialidade, integridade de dados e autenticação do cliente e do servidor

(KUROSE, 2012). Na Figura 2.3, é possível observar a diferença entre o modelo TCP/IP tradicional e o que inclui TLS/SSL.

Figura 2.3 – Comparação entre o modelo TCP/IP e o modelo TCP/IP com TLS/SSL



Fonte: Slaviero (2011).

O HTTP é um protocolo sem estado que utiliza o TCP como protocolo da camada de transporte. O modo de operação padrão do HTTP faz uso de conexões persistentes, mantendo uma conexão aberta para transferir múltiplos objetos do servidor para um mesmo cliente até que um intervalo de *timeout* seja atingido, mas é possível configurar o cliente e o servidor para utilizar conexões não persistentes, onde a conexão é fechada após a transferência de um objeto (como um arquivo HTML ou uma imagem) e outra conexão precisa ser aberta para que uma nova transferência seja feita.

Um mecanismo empregado para que um servidor identifique um usuário é o uso de *cookies*. Este recurso do HTTP permite que o servidor inclua um número de identificação único (que é armazenado em uma base de dados) em uma mensagem de resposta ao cliente. O navegador armazena este valor recebido em um arquivo e passa a incluí-lo em novas mensagens do cliente destinadas ao *hostname* do servidor. O transporte seguro dos *cookies* e do conteúdo das mensagens HTTP é imprescindível para a realização de transações na internet. Para que elas sejam possíveis, o protocolo Hypertext Transfer Protocol Secure (HTTPS), que é a implementação do HTTP com TLS/SSL, é utilizado.

2.4 Servidores Web

O termo servidor Web se refere ao software responsável por oferecer páginas Web com seus arquivos associados a clientes que os solicitem através do protocolo HTTP e ao computador no qual este programa é executado, que deve ter uma alta disponibilidade para atender a múltiplos clientes continuamente. Segundo o site W3Techs, no dia 06 de setembro de 2017, o servidor Web Apache é o mais popular, sendo utilizado por 49% dos 10 milhões de sites considerados na pesquisa (W3TECHS, 2017).

Tipicamente, uma pilha de softwares que inclui um sistema operacional, um servidor Web, um Sistema de Gerenciamento de Banco de Dados (SGBD) e um interpretador de uma linguagem de script é instalada neste computador. Esta solução, amplamente utilizada por aplicações Web, deve ser analisada sob a perspectiva de segurança, já que uma vulnerabilidade explorada em qualquer um destes softwares pode expor dados sigilosos dos usuários ou permitir a execução de código malicioso nesta máquina.

A instalação padrão dos softwares desta pilha muitas vezes adiciona serviços que não são necessários para um servidor Web e, portanto, precisam ser removidos ou configura serviços importantes de maneira insegura, apenas com o objetivo de demonstrar seu funcionamento. Alguns exemplos são o serviço de File Transfer Protocol (FTP), que pode ser usado por um atacante para realizar o upload de arquivos maliciosos ao servidor, e serviços de acesso remoto configurados automaticamente com usuários e senhas padrão.

Aplicações Web em desenvolvimento costumam possuir uma grande quantidade de vulnerabilidades, então este processo deve ser realizado em um ambiente diferente do ambiente de produção. Para que isso ocorra, o sistema deve ser desenvolvido e testado localmente na máquina do desenvolvedor (ou em uma máquina virtual sendo executada em seu computador) configurada apropriadamente para este propósito.

Outra prática que deve ser adotada pelos responsáveis por administrar um servidor Web é manter estes softwares atualizados. Frequentemente, novas versões de um software corrigem vulnerabilidades existentes nas versões anteriores, então convém aplicar as novas atualizações mesmo que uma vulnerabilidade não tenha sido explorada e divulgada publicamente.

Com o objetivo de identificar comportamentos anômalos, o servidor deve ser auditado periodicamente. Os logs dos softwares devem ser habilitados e analisados para que, caso um ataque seja bem-sucedido, a vulnerabilidade explorada seja localizada, os danos provocados

sejam medidos e o atacante seja rastreado. Obtendo estas informações, é possível corrigir o erro que permitiu o ataque e tomar as medidas judiciais cabíveis.

A instalação de um Web Application Firewall (WAF) como o ModSecurity (MODSECURITY, 2017) pode proteger o servidor de ataques comuns a aplicações Web através da análise do tráfego HTTP. Estes kits de ferramentas contribuem para a segurança com monitoramento da aplicação em tempo real, registro do tráfego HTTP e filtragem de funcionalidades do HTTP aceitas. Ainda, é possível customizar seu comportamento especificando um conjunto de regras descritas em uma linguagem interpretada pelo módulo de segurança.

Para utilizar o protocolo HTTPS, um certificado digital deve ser instalado no servidor. Através dele, a autenticidade do servidor pode ser checada pelos clientes, que verificam se o certificado foi assinado por uma autoridade certificadora confiável e se o *hostname* do servidor está de acordo com o informado no certificado.

3 VULNERABILIDADES CRÍTICAS

Um sistema computacional pode ser visto como uma máquina de estados. A decisão a respeito de quais estados e transições são autorizados ou não é definida por uma política de segurança. Para que um ataque ocorra, é necessário que exista um estado vulnerável no sistema. Este estado vulnerável é um estado autorizado que serve como ponto de partida do ataque, permitindo que, através de transições de estado autorizadas, um estado não autorizado seja atingido.

Vulnerabilidades permitem que um usuário sem acesso ao sistema consiga acessá-lo ou que um usuário com acesso parcial possa acessar recursos ou funcionalidades que não lhe foram permitidos. Uma vulnerabilidade pode ser causada por um erro de programação, de configuração ou de operação. Um ataque é um método para explorar uma vulnerabilidade. O ataque é um programa ou um método de engenharia social usado por um atacante para atingir seu objetivo, tirando vantagem da vulnerabilidade (BISHOP, 1996).

Em 2013, a Open Web Application Security Project (OWASP), uma organização sem fins lucrativos focada em aprimorar a segurança de software, listou os dez riscos de segurança mais críticos em aplicações Web (OWASP, 2013). Este estudo classificou os riscos considerando a dificuldade de exploração, a prevalência da vulnerabilidade, a dificuldade de detecção e os impactos técnicos causados se um ataque for bem-sucedido. A Tabela 3.1 apresenta a classificação feita pela OWASP sobre a dificuldade de exploração e os impactos técnicos causados pelas vulnerabilidades estudadas neste trabalho.

Tabela 3.1 – Dificuldade de exploração e impacto técnico dos 4 riscos de segurança mais críticos em aplicações Web

Vulnerabilidade	*Exploração*	*Impacto Técnico*
Injeção	Fácil	Severo
Quebra de Autenticação e Gerenciamento de Sessão	Média	Severo
Cross-Site Scripting	Média	Moderado
Referência Insegura e Direta a Objetos	Fácil	Moderado

Fonte: OWASP (2013).

Um relatório emitido em 2017 pela empresa WhiteHat Security, que desenvolve soluções de segurança, analisou dezenas de milhares de sites e aplicações seguindo o método Dynamic Application Security Testing (DAST). As vulnerabilidades encontradas foram classificadas nos seguintes níveis de risco: crítico, alto, médio, baixo e observação (WHITEHAT SECURITY, 2017).

No ranking estabelecido na pesquisa realizada pela OWASP, os quatro riscos principais foram: falhas de Injeção, Quebra de Autenticação e Gerenciamento de Sessão, Cross-Site Scripting e Referência Insegura e Direta a Objetos. Segundo a análise feita pela WhiteHat Security, as quatro classes de vulnerabilidades críticas na maioria das aplicações foram: SQL Injection, Cross-Site Scripting, Cross-Site Request Forgery e Autorização Insuficiente.

Relacionando os dados das duas pesquisas, é possível resumir falhas de Injeção a vulnerabilidade mais comum deste tipo (SQL Injection) e resumir Autorização Insuficiente a Referência Insegura e Direta a Objetos, já que ambas se tratam da falha em evitar que um usuário acesse dados ou realize operações sem autorização. Com isso, um conjunto com as cinco vulnerabilidades mais críticas é formado, contendo: SQL Injection, Quebra de Autenticação e Gerenciamento de Sessão, Cross-Site Scripting, Cross-Site Request Forgery e Referência Insegura e Direta a Objetos.

3.1 SQL Injection

Structured Query Language (SQL) é uma linguagem utilizada para estruturar, gerenciar e recuperar dados armazenados em bancos de dados relacionais. Existem instruções para: criar e destruir bases de dados; criar, alterar e destruir uma tabela em uma base de dados; inserir, alterar e deletar registros em uma tabela; selecionar dados de uma base de dados (GROFF, 2009). Consultas SQL contém uma lista de colunas, a tabela que será acessada para buscar os dados e, opcionalmente, um filtro a ser aplicado sobre os registros retornados, com o seguinte formato:

```
SELECT <nomes das colunas separados por vírgula>
FROM <nome da tabela>
[WHERE <nome da coluna> <operador de comparação> <valor>];
```

Aplicações Web frequentemente realizam consultas dinâmicas ao banco de dados, que dependem de uma ou mais entradas do usuário para serem formadas. Um exemplo de situação em que isto ocorre é durante a autenticação de usuário. Durante o processo de login, um usuário precisa fornecer informações como seu nome de usuário e sua senha para acessar o sistema.

Supondo que exista uma tabela no banco de dados chamada ‘usuarios’ que contenha os campos ‘nome’ e ‘senha’ e que há um registro nesta tabela com os dados fornecidos por um usuário durante seu cadastro no sistema. Para verificar se o usuário possui ou não acesso ao sistema, é necessário realizar a seguinte consulta SQL que filtre os registros desta tabela que possuam o nome e a senha informados:

```
SELECT nome, senha
FROM usuarios
WHERE nome = 'Alice' AND senha = 'VO5rA5MD';
```

Se nenhum registro que contenha os valores informados pelo usuário for encontrado, o usuário não possui acesso ao sistema e sua tentativa de login deve falhar. Caso contrário, sua autenticação é realizada com sucesso. Entretanto, se um usuário malicioso inserir comandos da linguagem SQL ao invés de seu nome de usuário e sua senha e as entradas não forem tratadas adequadamente pelo sistema antes serem utilizadas na consulta, um ataque pode burlar o processo de login.

A vulnerabilidade de SQL Injection ocorre quando um texto inserido pelo usuário no sistema é interpretado pelo SGBD como um conjunto de instruções ao invés de dados. Se no processo de login o usuário inserir, ao invés de seus dados cadastrados, uma sequência válida de comandos SQL que ignore a verificação do nome e da senha acima para fazer com que a consulta retorne sempre pelo menos um registro do banco e a aplicação Web utilizar o primeiro registro retornado na consulta para autenticar o usuário, o atacante pode acessar o sistema se passando pelo usuário que corresponde a este registro.

Para realizar um ataque, ao invés de seu nome, o usuário poderia inserir o caractere `'` para fechar a cadeia de caracteres aberta no código da aplicação, seguido de um comando SQL para realizar a operação de disjunção lógica entre a cadeia de caracteres vazia resultante e alguma condição que seja sempre verdadeira (por exemplo, o teste `1 = 1`). Para ignorar o restante da linha, os caracteres `--` seguidos de um espaço em branco podem ser inseridos, já que, segundo a linguagem SQL, esta combinação de caracteres define que os demais caracteres até a quebra de linha compõem um comentário. A consulta resultante é a seguinte:

```
SELECT nome, senha
FROM usuarios
WHERE nome = '' OR 1 = 1 -- ' AND senha = '';
```

3.2 Quebra de Autenticação e Gerenciamento de Sessão

Vulnerabilidades que permitam a quebra de autenticação e gerenciamento de sessão resultam na possibilidade de que um usuário consiga assumir a identidade de outro comprometendo senhas, chaves e ids de sessão. Os métodos de exploração destas vulnerabilidades podem ser específicos para cada aplicação, mas existem erros comuns que podem permitir que um ataque deste tipo seja realizado.

O armazenamento de senhas em texto puro, apesar de ser simples de implementar, é um erro que pode causar o comprometimento de todas as contas dos usuários do sistema. Qualquer pessoa com acesso ao banco de dados pode realizar uma consulta e descobrir, a partir do nome do usuário ou de qualquer outra informação conhecida que esteja contida em seu registro, sua senha para acessar o sistema assumindo sua identidade. Esta exploração pode ser realizada por uma pessoa que tenha acesso permitido ao banco de dados ou por um usuário que obtenha este acesso através de outra técnica, como a apresentada na seção anterior.

A funcionalidade de recuperação de senha é necessária para que um usuário cadastrado que esqueceu sua senha consiga acessar o sistema novamente. Nesta funcionalidade, uma abordagem comum é utilizar perguntas de segurança, que são respondidas pelo usuário durante o seu cadastro, e comparar as respostas fornecidas no formulário de recuperação com as armazenadas no banco de dados para permitir a definição de uma nova senha. Se as respostas para as perguntas escolhidas estiverem disponíveis publicamente, como “Qual é a sua data de aniversário?” ou “Qual é o seu código postal?” (CAMPANILE, 2008), um atacante que pesquise por estas informações e as encontre é capaz de definir uma nova senha para acessar a conta do usuário.

3.3 Cross-Site Scripting

Em aplicações Web que geram as páginas dinamicamente, o conteúdo das páginas é construído por scripts executados pelo servidor e depende de entradas fornecidas pelo usuário.

Estas entradas são enviadas ao servidor através de arquivos enviados ou formulários submetidos pelo cliente (utilizando os métodos POST e GET do protocolo HTTP). Ainda, o Document Object Model (DOM), modelo que representa a página carregada, pode ser modificado dinamicamente no cliente por código JavaScript, que é executado pelo navegador. Utilizando o conjunto de tecnologias Asynchronous JavaScript and XML (AJAX), o cliente é capaz de realizar requisições de dados assíncronas ao servidor sem que a página tenha que ser recarregada.

Os nomes dos campos do formulário (definidos pelo atributo *name* de cada elemento *input* no código HTML da página) e os dados fornecidos pelo usuário em cada campo são associados em uma *query string*, que codifica os campos em pares nome-valor. Se o formulário for submetido através do método GET, a *query string* é concatenada ao fim do Uniform Resource Locator (URL) da página (sendo visível na barra de endereços do navegador). Se o método POST for utilizado, o conjunto de pares nome-valor é submetido no corpo da requisição HTTP e o URL não é modificado.

A vulnerabilidade de Cross-Site Scripting ocorre quando códigos de scripts maliciosos são inseridos em páginas Web e executados pelos navegadores dos usuários que acessem estas páginas. A execução de scripts no navegador pode redirecionar o usuário para outro site de interesse do atacante, roubar *cookies* de sessão se o usuário estiver autenticado no site ou alterar a página para exibir um formulário falso com o objetivo de induzir o usuário a fornecer sua senha. As vulnerabilidades de XSS podem ser classificadas em dois tipos: *reflected* (ou *non-persistent*) e *persistent*.

3.3.1 Reflected

A falha chamada de *reflected* XSS ocorre quando a entrada do usuário modifica o conteúdo da próxima página carregada e o servidor não filtra esta entrada adequadamente. Supondo que o usuário acesse a funcionalidade de busca de um sistema (onde o termo inserido pelo usuário é procurado em todos os artigos armazenados pela aplicação, por exemplo) e que o campo de busca preenchido pelo usuário esteja contido em um formulário que utiliza o método GET, a página com os resultados da busca exibirá o termo procurado junto com os resultados e terá em seu URL a *query string* com o valor inserido pelo usuário.

A *tag* <script> da linguagem HTML indica aos navegadores que o conteúdo do elemento deve ser interpretado como código JavaScript. Se, ao invés de um texto comum, o campo de busca for preenchido com uma *tag* deste tipo contendo o código do atacante, ao

carregar a próxima página de resultados da busca (cujo URL contém a *query string* com o código malicioso), o navegador executará o código malicioso submetido. Um atacante pode enviar este URL para suas vítimas, que, ao acessarem o *link*, executarão o código malicioso em seus navegadores.

3.3.2 Persistent

Persistent XSS se manifesta quando o código malicioso é armazenado no banco de dados do servidor, que, ao receber a requisição de um cliente pela página que exibe a informação armazenada, insere o código malicioso (que será executado pelo cliente) no lugar onde deveriam haver apenas dados. A exploração deste tipo de vulnerabilidade afeta todos os usuários que acessarem a página. Ao contrário de *reflected* XSS, para que este tipo de vulnerabilidade seja explorada, não é necessário que a vítima do ataque clique em nenhum *link* enviado pelo atacante.

Tipicamente em blogs, wikis e fóruns online, existem campos de texto onde os usuários podem realizar comentários. O texto destes comentários pode ser destacado (em negrito, itálico, alterando o tamanho da fonte, etc.) se o componente utilizado para este campo do formulário for um editor What You See Is What You Get (WYSIWYG). Quando estes componentes são utilizados, a tarefa de impedir a execução de código malicioso se torna mais complexa, já que a formatação do texto é realizada através da adição de *tags* válidas do código HTML (por exemplo, a *tag* <b> para destacar um texto em negrito).

3.4 Cross-Site Request Forgery

A vulnerabilidade de Cross-Site Request Forgery ocorre quando requisições HTTP são criadas por um atacante, que as envia se passando por um usuário já autenticado no sistema vulnerável. Estas requisições incluem informações válidas, como cookies da sessão do usuário, e, portanto, são aceitas pela aplicação Web. As requisições forjadas são realizadas quando a vítima acessa uma página Web em outro sistema (tipicamente de posse do atacante), enquanto está autenticada no sistema alvo do ataque.

O sistema mantido pelo atacante pode realizar as requisições durante o carregamento da página ao utilizar um *link* para uma operação de transação no sistema alvo como valor do atributo src em uma *tag* <img> da linguagem HTML. O uso deste atributo faz com que o

navegador envie uma requisição HTTP utilizando o método GET para o endereço especificado (que contém a *query string* adulterada com os valores necessários para realizar a transação) com o objetivo de buscar a imagem a ser exibida, mas, neste caso, ao invés de buscar por uma imagem, a transação não autorizada será realizada quando a requisição for enviada.

Se o formulário do sistema alvo for submetido através do método POST, a técnica citada anteriormente não será efetiva. Entretanto, uma alternativa que pode ser empregada para realizar o ataque neste caso é a criação de um formulário na página do sistema mantido pelo atacante com os mesmos campos que o formulário existente no sistema vulnerável, mas cujos valores são pré-definidos. Ao fim do carregamento da página, através da execução de código JavaScript, o formulário pode ser submetido sem que o usuário realize nenhuma ação.

3.5 Referência Insegura e Direta a Objetos

Em uma aplicação que possui múltiplos usuários com diferentes permissões, quando um usuário autenticado é capaz de alterar o valor de um parâmetro e, com isso, consegue acessar um objeto que não deveria poder acessar (de acordo com a política de segurança estabelecida), existe uma falha de referência insegura e direta a objetos no código do sistema. O objeto acessado pode ser um arquivo armazenado no servidor ou um registro de uma tabela do banco de dados.

Quando o controle de acesso é violado através da exploração desta vulnerabilidade, o atacante pode modificar ou excluir informações essenciais para o funcionamento correto da aplicação ou ler dados sensíveis de outros usuários. Por exemplo, em uma aplicação Web que possui uma página onde um usuário pode alterar os dados do seu cadastro no sistema, se o URL desta página for formada pelo identificador único do usuário, um atacante autenticado pode acessar a página do seu cadastro, alterar o URL manualmente para substituir seu identificador pelo de outro usuário e modificar os dados do cadastro deste usuário.

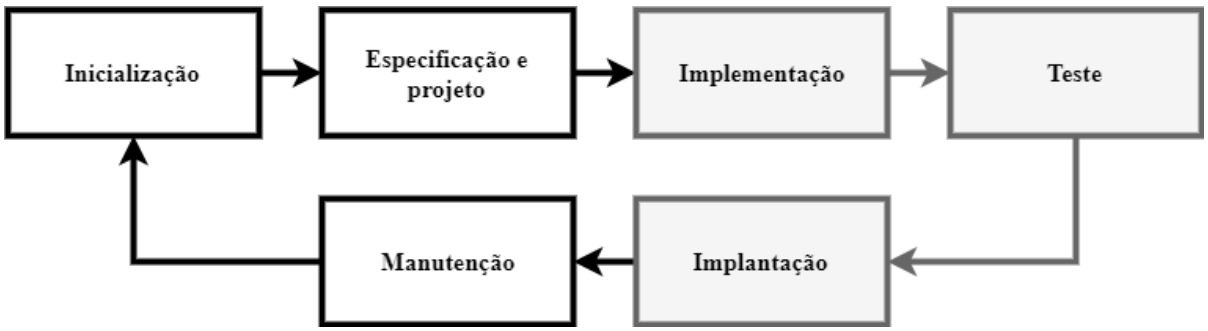
No caso de um sistema que permita que os usuários façam *upload* de arquivos, se os arquivos de todos os usuários forem armazenados no mesmo diretório do servidor e o URL utilizada para realizar o *download* de um arquivo for formada pelo nome do arquivo, uma alteração no URL pode permitir que o atacante realize o *download* de arquivos de outros usuários. Ainda, alterando o URL, um usuário comum pode acessar funcionalidades do sistema que só devem ser acessadas por um usuário com privilégios de administrador, como

uma página para criar novos usuários ou excluir usuários existentes no sistema, cujo *link* para acessá-la pela interface do sistema só é visível pelo administrador, mas pode ser deduzido por um atacante.

4 MEDIDAS DE PREVENÇÃO

O ciclo de vida de desenvolvimento de software é composto pelas fases de inicialização, especificação e projeto, implementação, testes, implantação e manutenção (Figura 4.1). Os desenvolvedores devem se preocupar com aspectos de segurança ao longo de todo o desenvolvimento do sistema, mas principalmente durante as etapas de implementação, testes e implantação (ANTUNES, 2012).

Figura 4.1 – Versão simplificada do ciclo de vida de desenvolvimento de software



Fonte: Antunes (2012, p. 67).

Na fase de implementação, boas práticas de programação para evitar vulnerabilidades incluem a validação de entradas e saídas, identificação de caracteres maliciosos e uso de comandos parametrizados. Programadores podem negligenciar aspectos de segurança por falta de conhecimento a respeito das vulnerabilidades introduzidas no sistema e das medidas que podem ser tomadas para preveni-las ou por acreditarem que a responsabilidade de tornar o sistema mais seguro é de outra pessoa, mas o código desenvolvido por eles é o principal alvo dos atacantes.

Técnicas como testes de intrusão, análise dinâmica ou estática e detecção de anomalias podem ser empregadas para identificar vulnerabilidades e corrigi-las durante a etapa de testes. Ferramentas de testes automatizados apresentam baixa cobertura de vulnerabilidades ou alto número de falsos positivos (FONSECA, 2007), então o emprego destas técnicas demanda um grande esforço dos desenvolvedores, que tipicamente recebem maior incentivo para testar requisitos funcionais da aplicação do que requisitos de segurança.

Mecanismos de detecção de ataques, chamados de Intrusion Detection Systems (IDSs) ou WAFs, podem ser adicionados quando os sistemas se tornam disponíveis para uso na etapa de implantação. O funcionamento dos mecanismos se baseia na identificação de variações de um comportamento aprendido através de detecção de anomalias ou de assinaturas. Entretanto, o custo adicional causado pela inclusão destes mecanismos pode degradar o desempenho do

sistema. Além disso, os resultados obtidos podem apontar para ameaças inexistentes na aplicação, assim como acontece com as ferramentas de testes automatizados.

Para a maioria dos projetos, a etapa de testes possui menor investimento do que a etapa de implementação e existem poucas pessoas responsáveis pela etapa de implantação. Grande parte dos desenvolvedores participa ativamente da etapa de implementação, porque ela está diretamente associada a funcionalidade do sistema. Adicionalmente, quanto antes uma vulnerabilidade for identificada, menor é o custo de corrigi-la. Por isso, as medidas de prevenção estudadas neste trabalho devem ser empregadas durante a etapa de implementação.

4.1 Gerenciamento Seguro de Credenciais

As senhas das contas dos usuários devem ser protegidas para que, no caso de um ataque bem-sucedido ao servidor Web, o atacante não seja capaz de acessar o sistema utilizando as credenciais de um usuário legítimo. As senhas não devem ser armazenadas em texto claro, mas uma função *hash* criptográfica deve ser utilizada, que recebe como entrada a senha do usuário anexada a um valor gerado randomicamente chamado de *salt*, e o valor resultante desta função deve ser armazenado, juntamente com o *salt*.

Dada uma entrada de tamanho arbitrário, a função *hash* gera uma cadeia de bits de tamanho fixo, de forma que seja inviável descobrir o valor de entrada conhecendo apenas o resultado da função. Para uma mesma entrada, a função sempre retorna o mesmo valor de saída. Portanto, se apenas a senha do usuário for utilizada como entrada da função e existirem dois ou mais usuários com a mesma senha, seus valores armazenados pelo sistema serão os mesmos.

Ataques de busca por força bruta, onde todas as possibilidades de entrada são testadas, ou de *rainbow tables*, que contém valores de funções *hash* pré-calculados e as entradas que os geraram armazenados em uma tabela, podem ser realizados se o armazenamento for feito desta forma e um atacante obtiver acesso ao banco de dados. Com o objetivo de evitar estes ataques, o *salt* é utilizado. Quando a autenticação é necessária, o valor de *salt* salvo durante o cadastro é anexado a senha inserida pelo usuário, a função *hash* é calculada e o valor resultante é comparado com o armazenado no banco de dados. Se os valores forem iguais, a autenticação é completada com sucesso.

Tipicamente, sistemas Web que permitem o cadastro de usuários oferecem a funcionalidade de recuperação de senha para que um usuário que esqueceu sua senha possa

acessar novamente sua conta sem a intervenção do administrador do sistema. Nesta situação, o uso de perguntas de segurança é conveniente, mas não é suficiente. As respostas para as perguntas de segurança frequentemente podem ser descobertas por um atacante através de uma pesquisa pelo nome da vítima em ferramentas de busca, já que a quantidade de informações pessoais publicadas em redes sociais é crescente, ou através de técnicas de engenharia social.

Uma abordagem mais segura para implementar esta funcionalidade consiste em gerar um código aleatoriamente, com uma quantidade mínima de caracteres e um tempo de expiração definidos, e enviá-lo para o usuário através de um canal alternativo, que tenha sido definido durante o preenchimento do seu cadastro no sistema. Este canal alternativo pode ser um e-mail de confiança ao qual o usuário tem acesso ou um número de telefone celular que seja capaz de receber mensagens de texto.

O código recebido pelo canal alternativo deve ser informado pelo usuário ao sistema para que sua nova senha seja definida. Após a definição da nova senha ou o tempo de expiração do código ser atingido, a aplicação não deve mais permitir que o código seja utilizado para redefinir a senha do usuário. Este procedimento deve ser realizado para que, caso um atacante obtenha este código no futuro, ele não seja capaz de reutilizá-lo.

Quando um atacante obtém acesso a uma lista de valores de *hash* gerados a partir de senhas de usuários de uma base de dados e conhece a função de *hash* empregada pela aplicação, ele possui tentativas ilimitadas para testar diversas senhas em sua máquina com o objetivo de descobrir qual senha gera o valor de *hash* correspondente ao armazenado para cada usuário. Se a senha de um usuário for descoberta desta forma, o atacante pode acessar o sistema utilizando a conta do usuário atacado e, caso o usuário utilize a mesma senha em mais de um serviço, outras contas da vítima também poderão ser acessadas indevidamente.

Para dificultar ataques de força bruta ou ataques de dicionário sobre as senhas dos usuários dessa forma, a senha definida durante o cadastro de um novo usuário do sistema deve atender a uma política definida pelos desenvolvedores. Políticas comuns impõem um limite mínimo de caracteres e exigem o uso de letras maiúsculas, letras minúsculas, símbolos e dígitos. O objetivo destas restrições é maximizar o tempo necessário para que um algoritmo desenvolvido para descobrir as senhas consiga cumprir o propósito para o qual foi criado. Idealmente, este tempo deve aumentado até que se torne inviável para um atacante realizar este procedimento com sucesso.

Ainda que existam gerenciadores de senhas que geram senhas complexas, as armazenam utilizando criptografia, permitem sua recuperação através de uma senha mestre e

poupam o usuário de ter que lembrar da senha de acesso de cada serviço em que possui uma conta, muitos usuários ainda criam novas senhas a cada novo cadastro sem a ajuda de nenhuma ferramenta ou reutilizam a mesma senha em mais de um serviço.

Uma pesquisa realizada em (KELLEY, 2012) apontou que, diante das condições testadas, aplicar apenas a restrição de tamanho mínimo de 16 caracteres na definição de senhas fornece mais segurança contra um atacante com grande poder de processamento do que exigir que a senha contenha pelo menos 8 caracteres com letras maiúsculas, letras minúsculas, um símbolo e um dígito. O emprego de políticas na definição de senha deve considerar a facilidade para que o usuário seja capaz de se lembrar da senha cadastrada. Uma senha longa composta apenas por letras pode ser mais fácil de memorizar do que uma que contém dígitos e caracteres especiais.

4.2 Tratamento Seguro de Entradas e Saídas

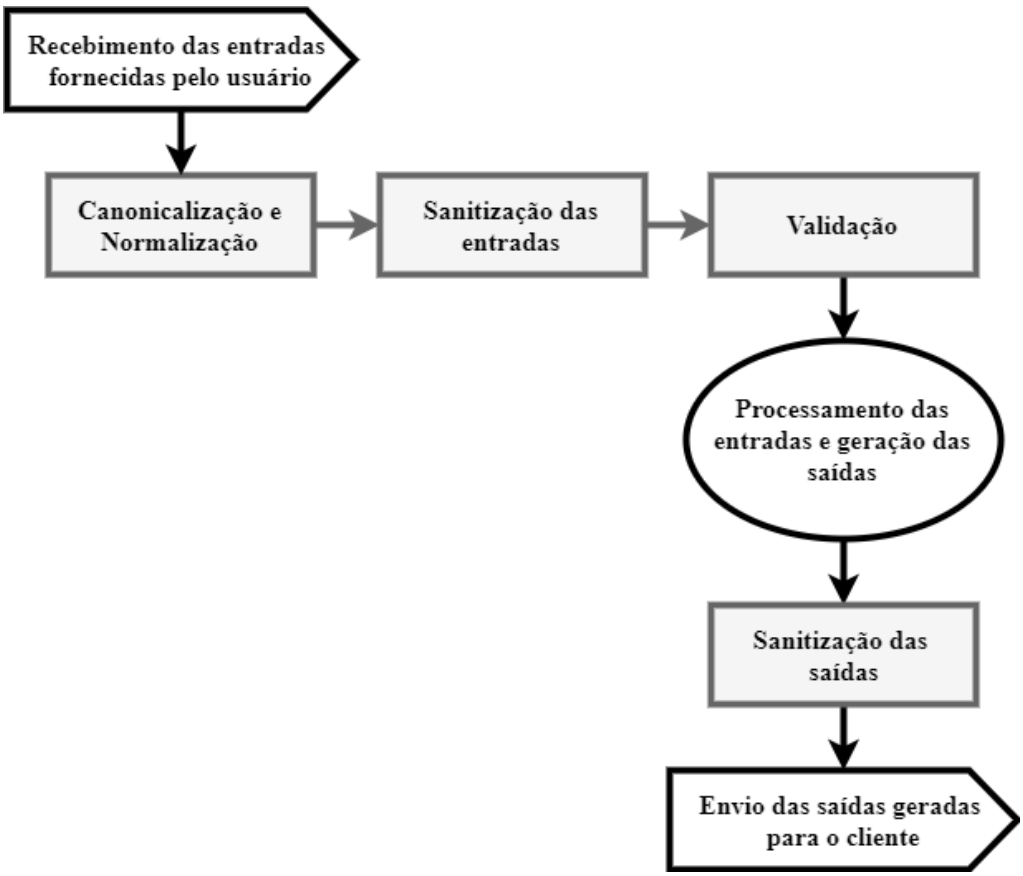
Dados inseridos pelo usuário não devem ser aceitos pela aplicação sem antes serem analisados. Qualquer entrada fornecida por um usuário deve ser considerada não-confiável. É necessário que estes dados sejam tratados pelas etapas de canonicalização e normalização, sanitização e validação antes de serem processados pela aplicação. A etapa de sanitização deve ser aplicada sobre as entradas e sobre as saídas geradas pelo sistema. A Figura 4.2 ilustra o processo de tratamento dos dados que deve ser realizado pelo servidor.

A canonicalização transforma uma entrada em sua representação mais simples sem que nenhuma informação se perca. Uma vez que o caminho para um arquivo no sistema de arquivos pode ser relativo ao diretório atual onde o programa está sendo executado, os caracteres ‘.’ e ‘..’ podem ser utilizados para representar o diretório atual e seu diretório pai, respectivamente. Se apenas o prefixo do caminho for verificado sem que o caminho seja canonicalizado, um atacante pode acessar um diretório indevidamente fazendo uso destas formas de representar o nome de um arquivo. Nomes de arquivos devem ser canonicalizados para assegurar que o caminho correto será verificado nas próximas etapas do tratamento de entradas.

O processo de normalização consiste em converter dados de entrada para sua forma mais simples permitindo que informações sejam perdidas. O padrão Unicode é adotado para representar textos por diferentes codificações de caracteres. Isso permite que uma mesma cadeia de caracteres possua diferentes representações. O Unicode também define regras para

normalização dos caracteres, que devem ser usadas para que cadeias de caracteres submetidas pelo usuário sejam verificadas adequadamente independente de qual representação de caracteres ele tenha utilizado.

Figura 4.2 – Etapas do processo de tratamento seguro de entradas e saídas



Fonte: Seacord (2015).

Dados de entrada e de saída podem ser alterados para se adequarem aos requisitos do subsistema ao qual estão sendo passados. Este procedimento é chamado de sanitização. A sanitização pode remover, substituir, codificar ou incluir caracteres de escape nos dados para evitar que ataques de injeção de código por parte de um usuário malicioso sejam efetivos. Usos comuns para a sanitização de dados incluem a substituição de caracteres reservados HTML por entidades de caracteres (representadas por nome ou número da entidade) e a remoção de *tags* HTML de uma cadeia de caracteres, a remoção de quebras de linha, tabulações e caracteres de espaço.

A validação de entradas é responsável por garantir que o valor de um campo de um formulário faz parte do conjunto de valores aceitos para aquele campo. Caso isso não seja verdade, nenhuma informação submetida pelo formulário é persistida no servidor e a página

para preenchimento do formulário é recarregada destacando os campos inválidos. Isso ocorre até que o usuário preencha todos os campos do formulário com valores válidos.

Em diversas situações, os componentes utilizados na página permitem que o usuário submeta valores que não são adequados para os campos do formulário e, mesmo que todos os componentes da interface aceitem apenas os valores válidos, eles podem ser alterados por um usuário que desabilite a execução de código JavaScript ou utilize um Web Proxy. Com a validação realizada pelo servidor, uma entrada que não pertença ao tipo esperado, não respeite um tamanho mínimo ou máximo, que seja vazia para um campo obrigatório ou que não atenda a outras restrições da aplicação é rejeitada.

4.3 Utilização de Tokens Únicos e Imprevisíveis

Com o objetivo de prevenir ataques do tipo CSRF, um *token* único e imprevisível deve ser gerado aleatoriamente pelo servidor e inserido em cada formulário que possa ser submetido por um cliente. Para uma sessão, é necessário gerar apenas um *token*, que é armazenado pelo servidor e é reutilizado em todos os formulários submetidos pelo usuário até a expiração da sessão. O *token* deve ser um valor longo, obtido a partir de um gerador de números aleatórios seguro, de forma que não possa ser previsto por um atacante. A aplicação Web é responsável por verificar a presença deste *token* nos formulários submetidos e sua validade. Se a validação do *token* falhar, a ação requisitada pelo cliente deve ser negada pelo servidor.

Para formulários que utilizem o método POST, um campo oculto que tem o *token* como valor é incluído. Quando o método GET é utilizado, o URL passa a conter o *token*. A desvantagem de utilizar o método GET é a exposição do *token*, que pode ser armazenado pelo histórico do navegador do usuário ou por arquivos de log de tráfego HTTP. Portanto, o uso desta medida de prevenção nos formulários que utilizam o método GET deve ser evitado. Assim, em formulários que alterem o estado do servidor, o uso do método POST é aconselhado.

Em aplicações que realizam chamadas AJAX, um *token* também deve ser utilizado para submeter informações ao servidor através do método POST. A abordagem de prevenção neste caso consiste em executar código JavaScript durante o carregamento de cada uma das páginas da aplicação para definir um cabeçalho HTTP customizado, que é chamado de X-CSRF-TOKEN. Este cabeçalho é submetido em todas as chamadas AJAX e contém o valor

do *token* da sessão gerado pelo servidor, que é inserido na geração da página. Quando a requisição é recebida pelo servidor, o valor do *token* contido no cabeçalho X-CSRF-TOKEN deve ser comparado com o armazenado na sessão do usuário para assegurar que o cliente submeteu o formulário de forma consciente.

4.4 Verificação de Controle de Acesso

O controle de acesso a funcionalidades e recursos de uma aplicação Web é necessário para que um usuário não realize operações não autorizadas. Para evitar que um atacante manipule o URL de uma página com o objetivo de acessar um objeto que sua conta de usuário não tem permissão para acessar, ao invés de referenciar diretamente o objeto pelo identificador de seu registro na base de dados ou pelo seu nome de arquivo no sistema de arquivos, a aplicação pode referenciar o objeto indiretamente, mapeando os identificadores dos objetos acessíveis pelo usuário autenticado no sistema em números sequenciais ou aleatórios, que podem ser exibidos ao usuário sem a exposição de informações sobre os registros no banco de dados ou nomes de arquivos.

A autorização determina quais são os direitos de acesso de um usuário após sua autenticação, definindo se o seu acesso a um recurso deve ser permitido ou negado pela aplicação. Existem modelos de controle de acesso consolidados na área de segurança da informação, que podem ser combinados ou utilizados individualmente para definir a política de controle de acesso de uma aplicação. A escolha desta política pode ser diferente de acordo com os tipos de usuários que utilizarão o sistema, então a análise dos melhores modelos a serem seguidos deve ser feita pelos desenvolvedores da aplicação.

O modelo de controle de acesso indicado para sistemas utilizados em empresas de grande porte é chamado de Role-Based Access Control, que restringe o acesso de um usuário ao sistema de acordo com seu cargo na empresa. Nestes sistemas, a definição dos cargos é facilitada por refletir a estrutura da organização. Seguindo este modelo, sempre que um funcionário mudar de cargo na empresa, é responsabilidade do administrador atualizar suas permissões no sistema de acordo com sua nova posição.

No modelo Discretionary Access Control, o usuário proprietário de um objeto é responsável por determinar quais usuários poderão acessá-lo, portanto o controle de acesso é baseado na identidade do usuário autenticado no sistema. Uma lista de controle de acesso é associada a cada objeto do sistema, que mantém a identificação dos usuários e seus

respectivos direitos de acesso sobre o objeto. Uma desvantagem deste modelo é a falta de controle que um usuário proprietário possui sobre o acesso ao objeto após seu compartilhamento com outro usuário.

O Mandatory Access Control classifica os objetos do sistema em níveis pré-definidos de sensibilidade. Neste modelo, não cabe ao usuário determinar o nível de acesso a um objeto, mas sim ao administrador do sistema. Sua utilização é comum em aplicações governamentais ou militares, onde dados sensíveis precisam ser protegidos. A sensibilidade de um objeto e a capacidade de acesso de um usuário são definidas através de rótulos, que contém um nível de segurança e uma lista de categorias de segurança. O nível de segurança é responsável por classificar a informação, enquanto as categorias de segurança definem a qual grupo a informação pertence. As relações entre os rótulos de usuários e de objetos determinam se um usuário pode acessar um objeto.

5 PROVA DE CONCEITO

Para demonstrar a importância do uso de medidas de prevenção durante o desenvolvimento de aplicações Web e identificar erros comuns cometidos pelos programadores, uma prova de conceito foi criada. A aplicação foi criada a partir de um *framework* de código aberto amplamente utilizado por empresas que desenvolvem sistemas Web. O sistema foi projetado de forma que as vulnerabilidades apresentadas neste trabalho pudessem ser exploradas e corrigidas.

O ambiente utilizado para executar a aplicação é virtualizado através de uma ferramenta chamada Vagrant (HASHICORP, 2017), que auxilia a criação e o gerenciamento de máquinas virtuais. O Vagrant suporta o uso de *boxes*, que são pacotes com ambientes virtuais pré-definidos. Existe um catálogo disponível publicamente com diversos destes ambientes. Neste trabalho, a versão 3.1.0 da *box* Laravel Homestead foi utilizada por incluir a pilha de softwares necessária para a execução da aplicação no servidor Web e por ser a *box* oficial sugerida pelos criadores do *framework* utilizado.

Alguns dos principais softwares incluídos na pilha fornecida pela Laravel Homestead (OTWELL, 2017) são:

- Sistema operacional: Ubuntu 16.04;
- Servidor Web: Nginx;
- SGBD: MySQL;
- Interpretador de linguagem de script: PHP 7.1.

Em um arquivo de configuração chamado Homestead.yaml, características da máquina virtual são definidas, como: endereço IP, quantidade de memória disponível, quantidade de CPUs, diretórios compartilhados com a máquina hospedeira, URL do site e nome da base de dados. Neste ambiente, o código da versão 5.5.14 do *framework* Laravel foi copiado para ser utilizado como ponto de partida no desenvolvimento da aplicação.

O projeto base do Laravel, que segue a arquitetura Model-View-Controller (MVC), contém a estrutura de diretórios e os arquivos necessários para exibir a página inicial do site. Em um arquivo de rotas (chamado web.php), os URLs do site são especificados e mapeados para métodos definidos nos controladores. O código dos controladores é escrito na linguagem PHP. Os métodos implementados nos controladores são responsáveis por atender as requisições HTTP recebidas pelo servidor Web.

Os métodos dos controladores implementam a lógica da funcionalidade requisitada e podem redirecionar o usuário a outro URL (comum para o método POST) ou retornar a página que será carregada pelo navegador do usuário (geralmente utilizado no método GET). O conteúdo de uma página é descrito utilizando um motor de templates chamado Blade. O Blade define uma linguagem que permite herdar templates (trechos de código que utilizados em mais de uma página, como o cabeçalho do site, por exemplo) e usar atalhos para estruturas de controle. Os arquivos das páginas devem ter a extensão .blade.php, já que são compilados em código PHP.

Uma interface de linha de comando chamada Artisan é oferecida para que o desenvolvedor possa criar novos comandos, que são programas escritos em PHP e podem ser executados pela Command-Line Interface (CLI), associados a aplicação desenvolvida ou executar comandos disponibilizados pelo Laravel. Com os comandos do Artisan e seus argumentos opcionais, esboços das classes que compõem a aplicação podem ser gerados, acelerando o processo de desenvolvimento e minimizando a quantidade de erros de programação.

Para inserir, alterar ou remover tabelas ou colunas em uma tabela no banco de dados, arquivos chamados de migrações devem ser criados. Através de código PHP orientado a objetos, estes arquivos podem definir, por exemplo, qual é o nome de uma tabela que deve ser criada no banco de dados e quais são as suas colunas usando métodos fornecidos pelo *framework*. Migrações podem ser criadas através de um comando do Artisan, editadas pelo usuário e executadas usando outro comando da CLI para realizar as modificações especificadas no banco de dados.

Após a criação das tabelas no banco de dados, é comum que seja necessário popular estas tabelas com registros no início do desenvolvimento da aplicação. Esta funcionalidade pode ser útil para pré-cadastrar usuários com senhas padrão automaticamente em sistemas que não permitem que visitantes realizem seus próprios cadastros ou para que desenvolvedores possam testar a aplicação com dados semelhantes aos de usuários reais. Existe um comando do Artisan para gerar esboços de classes para definir os valores dos registros das tabelas e outro para inseri-los no banco de dados.

O acesso aos registros do banco de dados é facilitado pelo uso de um Object-Relational Mapping (ORM) chamado Eloquent, que permite a inserção, a atualização e a remoção das informações persistidas por métodos de um objeto representando uma entrada em uma tabela. Os valores das colunas de uma tabela são acessíveis pelos atributos do objeto. É tarefa do desenvolvedor criar e atualizar, quando necessário, as classes que correspondem

as tabelas existentes no banco de dados (chamadas de modelos) para que a aplicação acesse os dados corretamente.

5.1 Aplicação Web

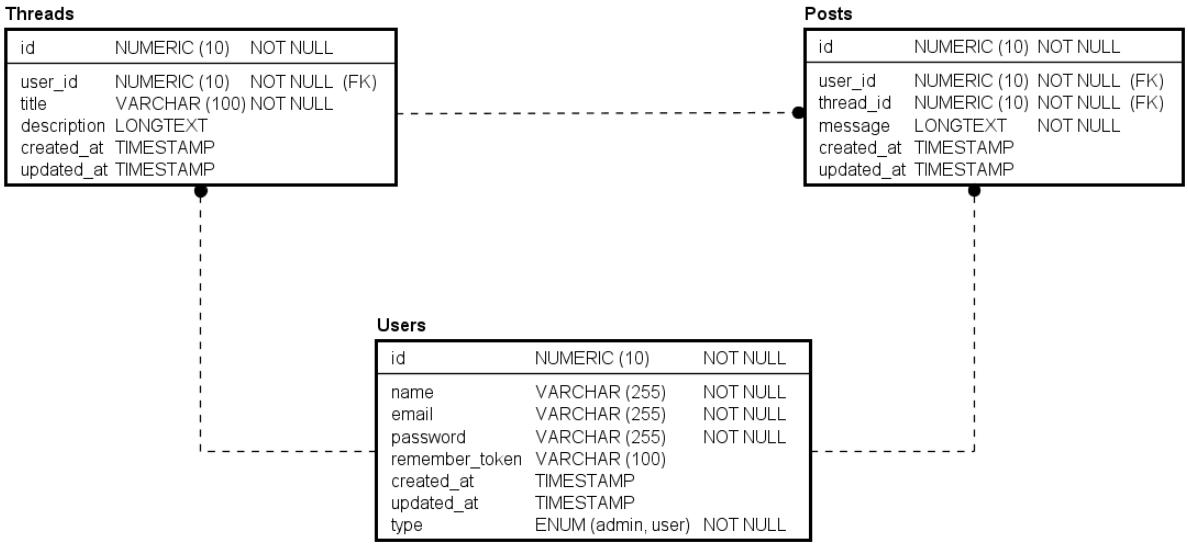
A aplicação Web criada para analisar os conceitos apresentados neste trabalho, chamada de Internet Forum, se trata de um fórum de discussão, onde usuários visitantes (sem cadastro) podem pesquisar por termos presentes em tópicos ou postagens feitos pelos usuários cadastrados. O fórum é composto por um conjunto de tópicos que contém um conjunto de postagens associadas. Todos os usuários cadastrados no fórum são capazes de realizar pesquisas, criar, editar e excluir seus próprios tópicos e postagens

Existe um tipo de usuário cadastrado que possui permissões adicionais. Usuários do tipo administrador podem excluir qualquer usuário, tópico ou postagem e editar as informações de qualquer usuário cadastrado, sendo capazes de redefinir a senha de um usuário ou torná-lo administrador também. Assim, existem três tipos de usuários para este sistema: visitante, usuário cadastrado comum e administrador. Qualquer usuário visitante é capaz de se registrar inserindo seu nome (que será exibido ao lado de seus tópicos e postagens), e-mail e uma senha (usados para realizar o login no sistema).

O Diagrama Entidade-Relacionamento na Figura 5.1 representa as estruturas de informação criadas no banco de dados através de migrações. Neste diagrama, é possível identificar as relações entres as tabelas (pelas chaves estrangeiras), os tipos dos campos das tabelas e a obrigatoriedade do preenchimento destes campos. Através das relações é possível observar que um usuário pode ser o autor de diversos tópicos ou postagens e que toda postagem pertence a um tópico.

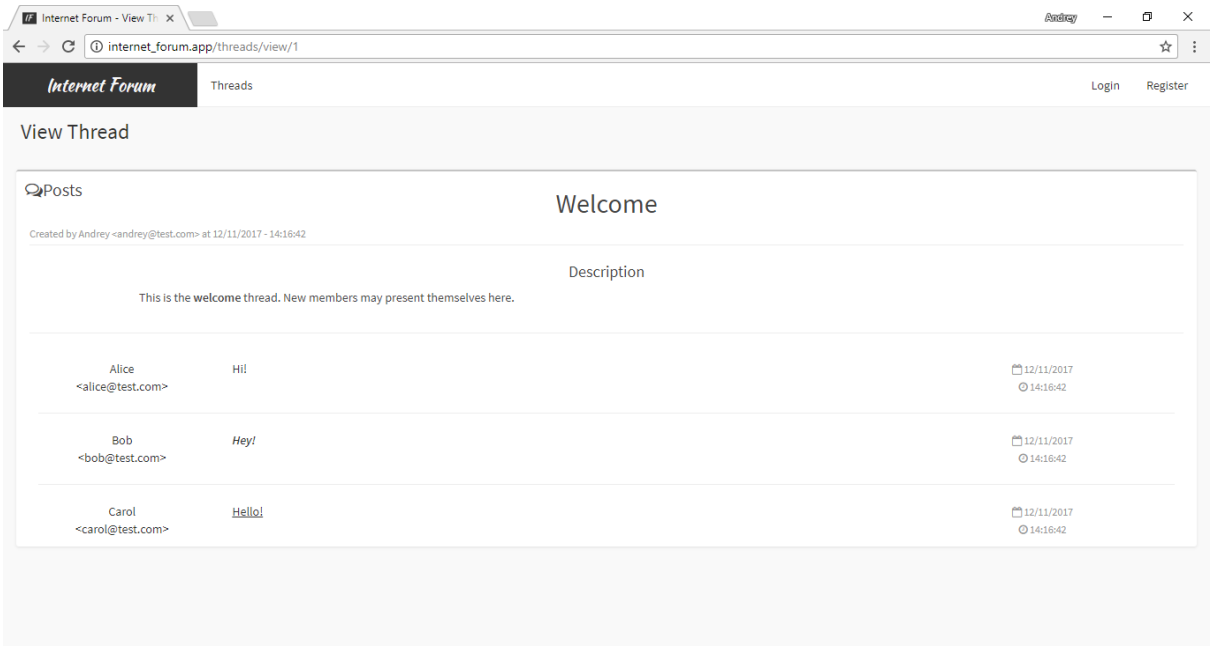
O recurso fornecido pelo *framework* para popular as tabelas do banco de dados automaticamente foi utilizado para iniciar o sistema com dados que permitam a reprodução dos experimentos descritos nas próximas seções. Estes dados incluem quatro usuários registrados com senhas padrão (um administrador e três usuários comuns), um tópico criado pelo administrador e três postagens neste tópico (uma feita por cada um dos outros usuários). A Figura 5.2 exibe a página de visualização deste tópico segundo o ponto de vista de um usuário visitante.

Figura 5.1 – Diagrama Entidade Relacionamento do Internet Forum



Fonte: Próprio autor.

Figura 5.2 – Visualização de um tópico por um usuário visitante



Fonte: Próprio autor.

5.2 Exploração das Vulnerabilidades

Durante o desenvolvimento do sistema Web apresentado, foi possível identificar casos em que a solução mais intuitiva ou que permite que as funcionalidades sejam implementadas mais rapidamente pode levar a introdução de vulnerabilidades de segurança na aplicação.

Nesta seção, a exploração de algumas das vulnerabilidades críticas apresentadas no Capítulo 3 é feita no sistema desenvolvido para exemplificar as técnicas que podem ser utilizadas por um atacante e demonstrar os riscos existentes caso um ataque seja realizado com sucesso.

5.2.1 SQL Injection no Processo de Login

Para tornar a exploração desta vulnerabilidade possível, os métodos de login e de registro no sistema fornecidos por padrão pelo Laravel foram sobrescritos. Na aplicação modificada, quando um usuário se registra no sistema, o valor armazenado no campo de senha da tabela de usuários no banco de dados, que guarda o *hash* da senha do usuário, foi calculado utilizando a função crypt() da linguagem PHP. Para esta função, além da senha fornecida pelo usuário, um valor de *salt* fixo foi utilizado. O prefixo do *salt* determina qual algoritmo de cifragem é utilizado pela função. O *salt* definido foi escolhido para que a função utilize o algoritmo Blowfish (SCHNEIER, 2017).

O login modificado realiza uma verificação semelhante a apresentada no exemplo da Seção 3.1 para autenticar os usuários. Uma consulta ao banco de dados é realizada para buscar o primeiro usuário encontrado na tabela de usuários que possua o endereço de e-mail inserido e cujo o *hash* da senha armazenado corresponda ao gerado a partir da senha submetida. Portanto, a senha inserida pelo usuário também é processada utilizando função crypt(), com o mesmo valor de *salt* utilizado no processo de registro.

Para realizar esta consulta, um construtor de consulta foi usado. Os construtores de consultas existentes no Laravel permitem formar e executar consultas na linguagem SQL apenas através do encadeamento de chamadas de métodos de um construtor de consulta. Ao invés de definir a consulta completa na linguagem SQL como uma cadeia de caracteres e fornecê-la como argumento para um método que a execute acessando o SGBD, neste paradigma, cada método chamado sobre o objeto compõe um trecho da consulta final.

Os construtores de consultas utilizam a vinculação de parâmetros para variáveis da extensão PHP Data Objects (PDO) para proteger a aplicação desenvolvida de ataques de SQL Injection. A vantagem de utilizar o construtor de consulta em detrimento ao uso direto de métodos de PDO está na simplicidade e na qualidade do código necessário para realizar uma consulta com o recurso fornecido pelo Laravel. A sintaxe dos construtores de consultas faz uso da orientação a objetos para definir a tabela consultada a partir de uma classe, fornece métodos para substituir os comandos mais comuns da linguagem SQL e auxilia na depuração

de erros cometidos na formação da consulta, já que a consulta é descrita na linguagem PHP ao invés da linguagem SQL.

Nos construtores de consultas, existem métodos que possuem o sufixo *raw*. Se uma cadeia de caracteres correspondente a um trecho de código SQL válido for fornecida como o único argumento para um destes métodos, esse código será inserido na consulta final realizada ao banco de dados sem nenhum tratamento de caracteres especiais. A implementação de login realizada utiliza o método whereRaw() do construtor de consultas, que recebe como único argumento o código que compara o endereço de e-mail de um usuário armazenado no banco de dados com o submetido pelo usuário no formulário. O código desta consulta é o seguinte:

```
$user = User::whereRaw('email = \'' . $request->get('email') . '\'')
   ->where('password', crypt($request->get('password'),
    '$2a$07$thissaltisreallyhardtoguess$'))
   ->first();
```

A consulta realizada para validar as credenciais do usuário busca o primeiro registro na tabela de usuários cujo endereço de e-mail corresponda ao submetido no formulário de login e cujo *hash* armazenado seja igual ao *hash* calculado a partir da senha submetida pelo usuário. Se um ou mais registros forem encontrados, o primeiro usuário retornado pela consulta é autenticado, mas, se nenhum registro for encontrado, a tentativa de login falha.

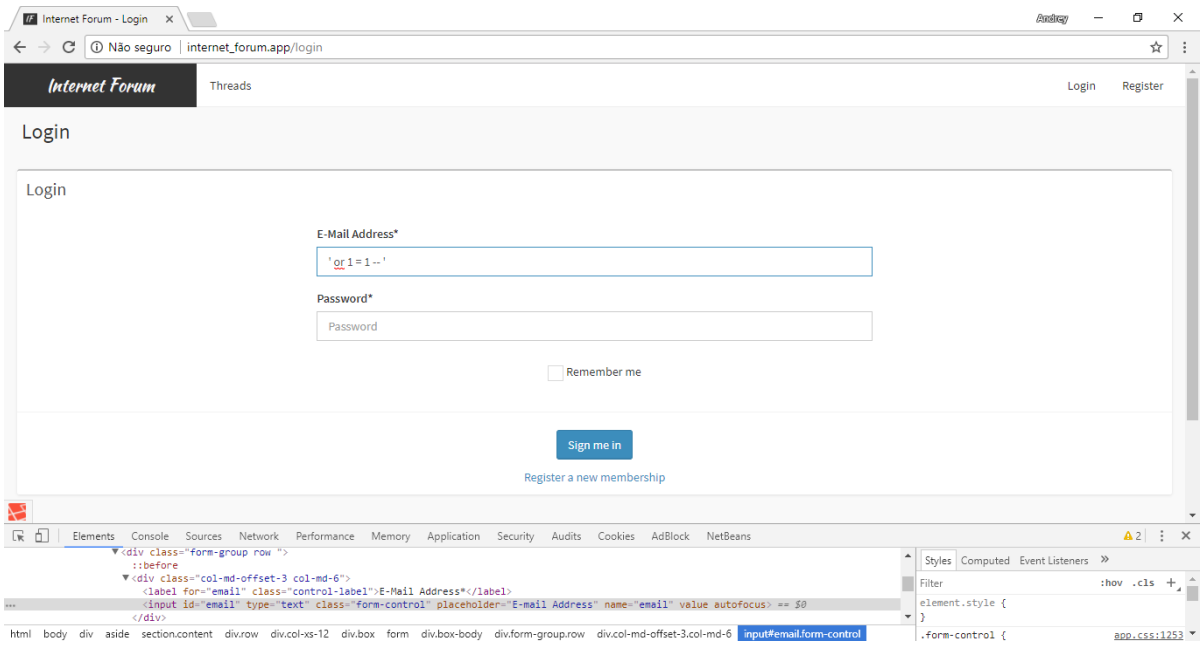
Esta implementação é insegura contra ataques de SQL Injection, já que a entrada fornecida pelo usuário no campo do endereço de e-mail do formulário de login é inserida diretamente na consulta que será executada pelo SGBD. Se, ao invés de seu e-mail, um atacante que não está registrado no sistema submeter código SQL neste campo, ele é capaz acessar o sistema como um usuário registrado.

Como o campo vulnerável neste caso é o de endereço de e-mail, o uso do atributo “type” com o valor “email” do elemento “input” no código HTML da página faz com que o navegador realize a validação da entrada, exigindo a presença do caractere ‘@’ no valor submetido. Entretanto, esta validação realizada apenas pelo cliente pode ser facilmente ignorada por um atacante fazendo uso do recurso “Inspecionar” oferecido pelos navegadores, editando o código HTML para trocar o valor deste atributo para “text”, que não exige a presença de caracteres especiais na entrada, por exemplo.

Após realizar este procedimento, o código malicioso pode ser submetido e será processado pelo servidor, que, nesta implementação, não valida a entrada do usuário

corretamente. A Figura 5.3 ilustra a entrada maliciosa fornecida por um atacante e o código HTML da página alterado para ignorar a validação do endereço de e-mail. A Figura 5.4 apresenta a consulta executada pelo SGBD na barra de depuração do Laravel e a tela exibida após a autenticação para o usuário do tipo administrador, que foi o primeiro registro retornado por esta consulta.

Figura 5.3 – Entrada maliciosa inserida durante o login para realizar um ataque de SQL Injection



Fonte: Próprio autor.

Figura 5.4 – Tela exibida ao usuário administrador após o login bem-sucedido

Fonte: Próprio autor.

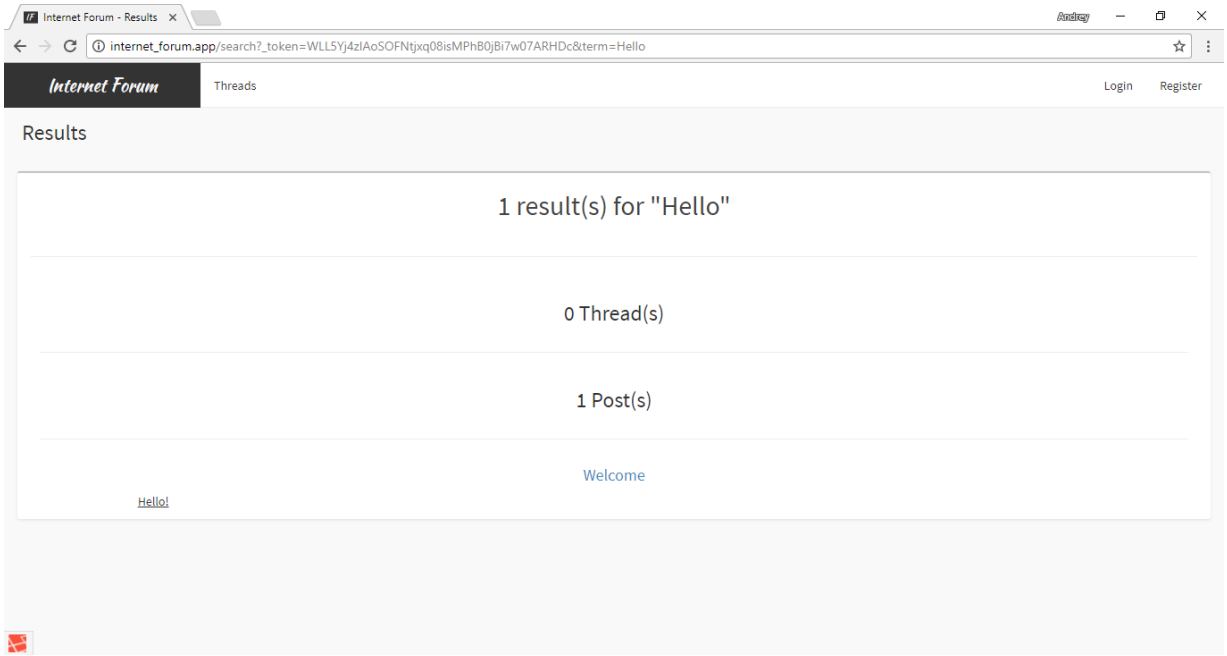
5.2.2 XSS na Pesquisa de Termos em Postagens e Tópicos

O motor de templates Blade é usado para auxiliar no desenvolvimento do código HTML das páginas criadas pela aplicação e enviadas ao cliente. Para páginas geradas dinamicamente, é comum que seja necessário exibir o valor de uma variável da linguagem PHP (que foi processado pela lógica de um controlador) para compor o código HTML final da página. A sintaxe do Blade permite que isso seja feito da seguinte forma:

```
{!! <variável da linguagem PHP> !!}
```

Este recurso foi utilizado na funcionalidade de busca por termos presentes em postagens ou tópicos dentro do fórum desenvolvido. Quando o usuário insere o termo e clica no botão de pesquisa, uma chamada HTTP GET é realizada. O servidor processa a requisição, realizando a busca pelas postagens e tópicos no controlador desta funcionalidade e cria a página que deve ser exibida a seguir. Para formar esta página de resultados, a variável que contém o termo pesquisado é utilizada. A Figura 5.5 apresenta o resultado obtido quando um usuário visitante pesquisa pelo termo ‘Hello’.

Figura 5.5 – Resultado da pesquisa pelo termo ‘Hello’



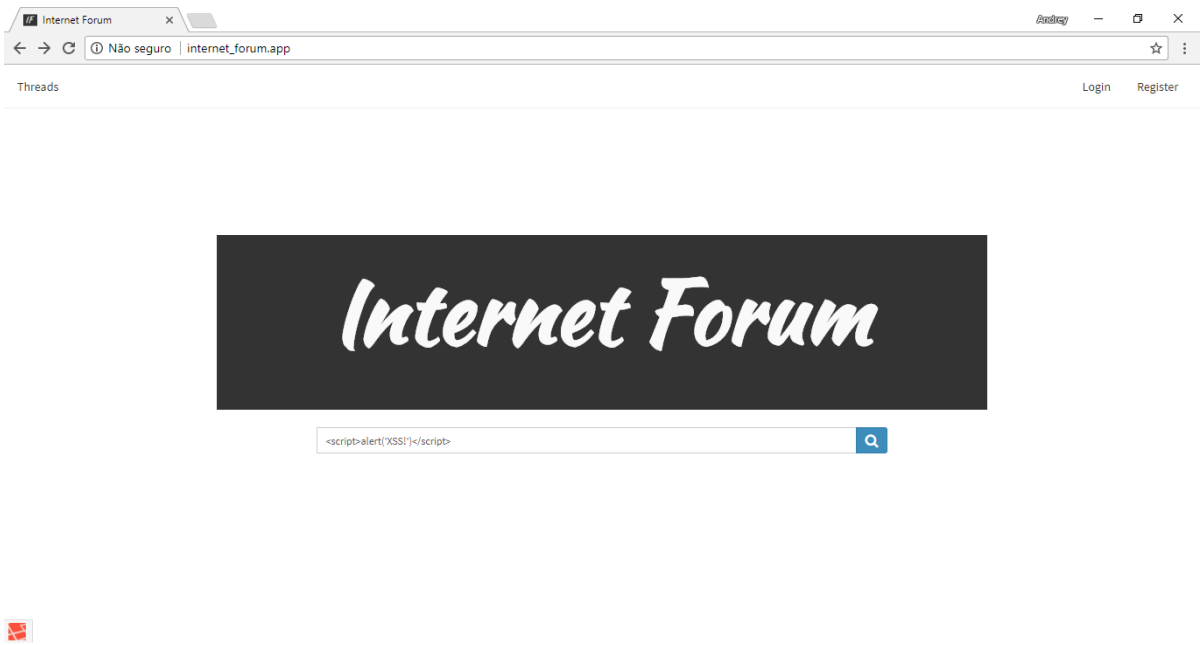
Fonte: Próprio autor.

Se o termo pesquisado fosse um trecho de código HTML, este código seria interpretado pelo navegador no carregamento da página. Através da *tag* <script> da

linguagem HTML, código JavaScript pode ser inserido na página e interpretado pelo navegador. Isso caracteriza um ataque de *reflected* XSS. A exploração desta vulnerabilidade permite que um atacante execute qualquer código JavaScript de sua autoria na máquina de um cliente legítimo que clique em um *link* malicioso. A Figura 5.6 ilustra a injeção de um código que exibe um alerta com a mensagem ‘XSS!’ no navegador da vítima.

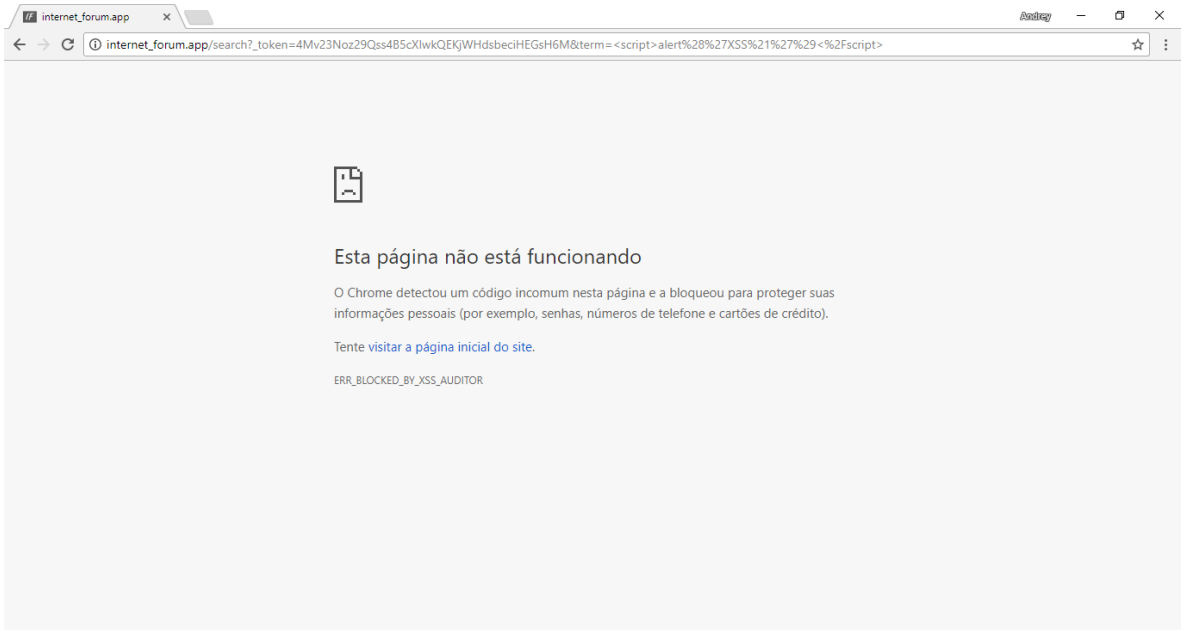
Ao realizar a consulta com esta entrada maliciosa, clicando no botão de pesquisa, a aplicação processa a requisição e insere a *tag* injetada pelo usuário no código HTML da página retornada ao cliente. O navegador Google Chrome (versão 62.0.3202.94) identificou o ataque e impediu o carregamento da página para proteger o usuário (Figura 5.7). Ao repetir o experimento no navegador Mozilla Firefox Quantum (versão 57.0), a página foi carregada e o alerta com a mensagem ‘XSS!’ foi exibido (Figura 5.8), comprovando que o ataque foi bem-sucedido.

Figura 5.6 – Injeção de código para exibir um alerta no navegador da vítima



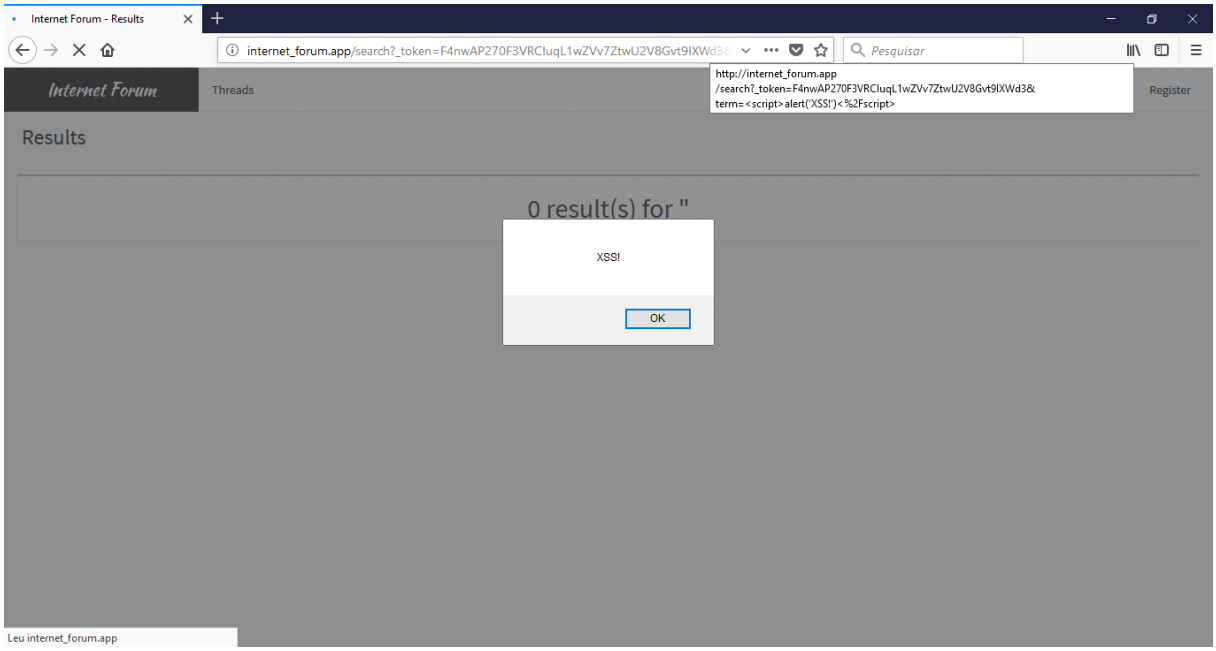
Fonte: Próprio autor.

Figura 5.7 – Página de resultados bloqueada pelo Google Chrome



Fonte: Próprio autor.

Figura 5.8 – Código malicioso executado durante o carregamento dos resultados no Firefox Quantum



Fonte: Próprio autor.

5.3 Correção das Falhas

As vulnerabilidades exploradas na seção anterior foram corrigidas utilizando corretamente os recursos oferecidos pelo *framework*. Esta seção detalha o processo de correção das falhas que viabilizaram os ataques apresentados, com o objetivo de avaliar a dificuldade de desenvolver as funcionalidades de forma segura em comparação com a forma mais simples descrita anteriormente, que não considera os aspectos de segurança da aplicação.

5.3.1 Login e Registro de Novos Usuários

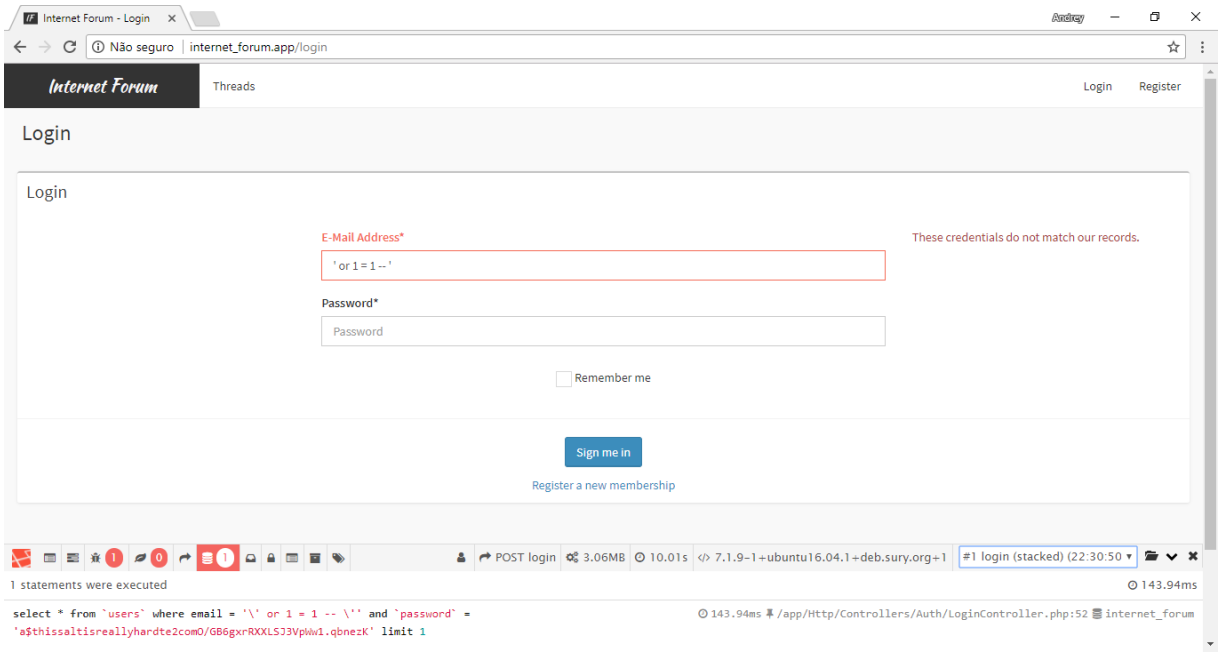
Na Subseção 5.2.1, um ataque onde um usuário sem registro no sistema consegue realizar o login com uma conta de administrador foi exemplificado. Este ataque foi possível devido a um conjunto de vulnerabilidades presentes nos processos de login e de registro de novos usuários. Nesta seção, o procedimento para corrigir a vulnerabilidade de SQL Injection será descrito e uma abordagem mais segura para registrar e autenticar usuários será apresentada.

Uma forma de impedir que código SQL submetido pelo usuário seja executado pelo SGBD é através da sanitização de entradas, como foi explicado na seção 4.2. O próprio método whereRaw(), utilizado no construtor de consulta para realizar o login e que permitiu a exploração da vulnerabilidade apresentada, permite realizar o tratamento correto de entradas (inserindo caracteres de escape) se for utilizado corretamente. A implementação deste método suporta o uso de um segundo argumento opcional, que é um vetor com os valores que devem ser sanitizados antes de serem inseridos na consulta.

Para corrigir esta vulnerabilidade, ao invés de utilizar o endereço de e-mail submetido pelo usuário diretamente na formação da cadeia de caracteres como primeiro argumento do método whereRaw(), o primeiro argumento deve ser composto por uma cadeia de caracteres que representa a comparação do campo email com uma variável representada pelo caractere ‘?’ e o segundo argumento deve ser um vetor com a variável a ser sanitizada. Aplicando esta alteração no código da funcionalidade de login, a entrada maliciosa não é tratada como código SQL pelo SGBD e a autenticação falha. A Figura 5.9 apresenta a falha na autenticação e a consulta que foi executada. O código corrigido que realiza esta consulta é o seguinte:

```
$user = User::whereRaw('email = ?', [$request->get('email')])
      ->where('password', crypt($request->get('password'),
      '$2a$07$thissaltisreallyhardtoguess$'))
      ->first();
```

Figura 5.9 – Falha na autenticação quando o endereço de e-mail é sanitizado corretamente



Fonte: Próprio autor.

Neste caso, o uso correto de um método foi suficiente para evitar a vulnerabilidade de SQL Injection, mas a funcionalidade de login deve considerar outros aspectos de segurança para que outros tipos de ataques ao sistema não sejam bem-sucedidos. O Laravel oferece por padrão controladores com as funcionalidades de login e de registro de usuários. Esta implementação deve ser utilizada sempre que possível e qualquer personalização necessária deve ser realizada apenas para incrementar o código existente nestes controladores com o objetivo de evitar a introdução de vulnerabilidades.

O registro de novos usuários na implementação padrão armazena o *hash* da senha cadastrada utilizando a função password_hash() da linguagem PHP, que gera um valor de *salt* aleatório a cada vez que é chamada. Assim, dois usuários com a mesma senha terão valores de *hash* diferentes armazenados no banco de dados. Esta abordagem é mais segura contra ataques que utilizem *rainbow tables*, como explicado na seção 4.1.

Para autenticar usuários no processo de login, o código fornecido pelo *framework* inclui um limite de tentativas de login falhas (5 tentativas, por padrão) por período de tempo

(1 minuto, se não for alterado). Se este limite for atingido, o usuário deve esperar alguns minutos para realizar uma nova tentativa. Esta medida é efetiva contra ataques de força bruta. A função utilizada para verificar se o valor de *hash* armazenado no banco de dados corresponde a senha submetida pelo usuário é a password_verify() da linguagem PHP e compara os valores de *hash* após a execução da consulta. Isso obriga o usuário a fornecer a senha correta para poder acessar o sistema.

5.3.2 Funcionalidade de Pesquisa

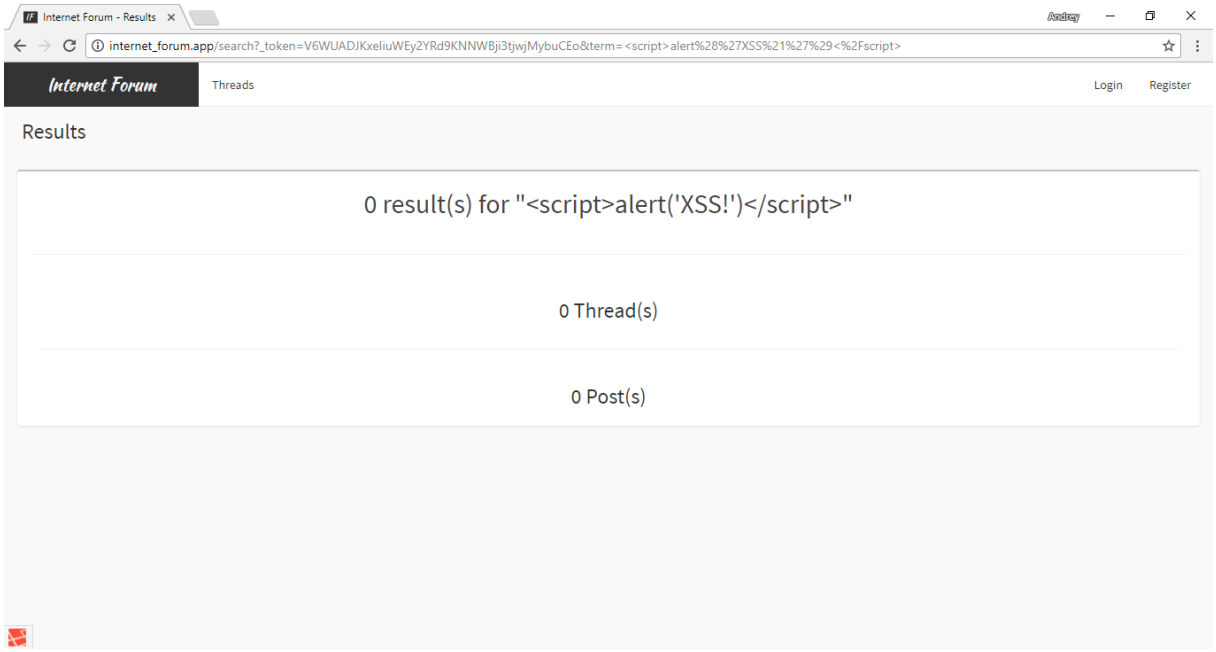
A demonstração do processo de exploração de uma vulnerabilidade na funcionalidade de pesquisa por termos inseridos pelo usuário no fórum foi realizada na Subseção 5.2.2. A vulnerabilidade do tipo *reflected* XSS foi causada pelo uso de uma entrada fornecida pelo usuário diretamente na formação da página de resultados sem a aplicação de nenhum tipo de sanitização sobre os dados antes de utilizá-los.

No caso desta funcionalidade, a presença de qualquer *tag* da linguagem HTML no termo procurado não deve ser interpretada pelo navegador como as demais *tags* que compõem a página de resultados, mas deve ser tratada como parte da cadeia de caracteres que define este termo. Para que as *tags* presentes na variável não sejam interpretadas como parte da página, o Blade suporta a seguinte sintaxe:

```
{{ <variável da linguagem PHP> }}
```

Utilizando esta sintaxe, o valor da variável é passado como parâmetro para a função htmlspecialchars() do PHP antes de ser inserido na página. Esta função transforma os caracteres especiais da linguagem HTML em entidades de caracteres. Por exemplo, o caractere ‘<’ é substituído por ‘<’ para que o navegador não interprete este caractere como o início de uma *tag*, mas sim como o símbolo matemático menor que. No caso da vulnerabilidade apresentada, a alteração no código que gera a página de resultados para utilizar este recurso do Blade ao exibir o termo pesquisado é suficiente para impedir que o ataque de *reflected* XSS seja efetivo. A Figura 5.10 demonstra que o código inserido pelo atacante não foi executado, mas foi tratado como uma cadeia de caracteres qualquer na construção da página.

Figura 5.10 – Página de resultados obtida quando o termo pesquisado é sanitizado corretamente



Fonte: Próprio autor.

6 CONCLUSÃO

Este trabalho tem como objetivo reforçar a importância de considerar aspectos de segurança durante o desenvolvimento de aplicações Web. Definições dos princípios de segurança e da arquitetura dos sistemas Web foram apresentadas, permitindo identificar os componentes que manipulam informações sensíveis e, portanto, devem ser protegidos contra usuários mal-intencionados. Algumas das tecnologias empregadas atualmente para tornar estes componentes mais seguros foram apresentadas.

Cinco vulnerabilidades críticas que têm afetado aplicações Web nos últimos anos foram detalhadas e exemplos de como os sistemas podem ser comprometidos por causa destas vulnerabilidades foram citados. As medidas de prevenção que podem e devem ser adotadas pelos desenvolvedores para evitar a introdução destas vulnerabilidades durante o projeto dos sistemas foram descritas.

Para prevenir as vulnerabilidades de SQL Injection e XSS, o tratamento seguro de entradas e saídas deve ser feito. A Quebra de Autenticação e Gerenciamento de Sessão pode ser evitada através do tratamento seguro de credenciais. Com a utilização de *tokens* únicos e imprevisíveis, os sistemas são protegidos de ataques de CSRF. A verificação de controle de acesso dos usuários é a solução que deve ser empregada para combater a Referência Insegura e Direta a Objetos.

Um fórum de discussão foi desenvolvido utilizando o *framework* Laravel, que possui código aberto e é amplamente adotado por empresas que projetam sistemas Web. Nesta aplicação, duas das vulnerabilidades descritas no trabalho foram exploradas e corrigidas. Os recursos oferecidos pelo Laravel são suficientes para evitar a exploração destas duas vulnerabilidades, mas, se forem utilizados da forma incorreta pelos desenvolvedores, podem introduzir brechas de segurança que permitam a um atacante acessar a conta do administrador do sistema sem as credenciais corretas ou executar código JavaScript de sua autoria no navegador de um cliente legítimo que clique em um *link* malicioso.

A primeira vulnerabilidade selecionada para realizar este experimento foi SQL Injection. Quando os construtores de consulta do Laravel são utilizados, métodos com o sufixo *raw* devem receber dois argumentos para prevenir esta vulnerabilidade: o primeiro descreve uma expressão na linguagem SQL (definida pelo desenvolvedor) e o segundo é um vetor de variáveis (entradas do usuário) que devem ser sanitizadas através da inserção de caracteres de escape antes de serem incluídas na consulta final.

O segundo ataque explorado e corrigido foi do tipo *reflected* XSS. Ao utilizar o Blade (motor de templates do Laravel), para exibir o valor de uma entrada do usuário na página gerada pela aplicação, a sintaxe que deve ser utilizada pelo desenvolvedor para evitar este tipo de ataque é {{ <variável da linguagem PHP> }}. Dessa forma, caracteres especiais presentes na entrada do usuário são transformados em entidades de caracteres.

A aplicação desenvolvida, disponível em (BLAZEJUK, 2017), pode ser utilizada por trabalhos futuros para explorar e corrigir outras vulnerabilidades de segurança que não foram exemplificadas neste trabalho. A vulnerabilidade de *persistent* XSS pode ser analisada nas postagens de usuários dentro dos tópicos. Um ataque do tipo CSRF pode ser estudado na submissão de formulários sem o *token* único e imprevisível (suportado pelo Laravel através da função csrf_token()). O acesso a funcionalidades do sistema não permitidas a usuários comuns pode ser investigado no arquivo de rotas da aplicação e no controle de acesso dos controladores que implementam as funcionalidades acessíveis apenas por usuários do tipo administrador.

Com a popularização da Internet e das aplicações Web, a segurança destes sistemas se tornou um tópico de extrema importância para que os dados dos usuários não sejam expostos. Os servidores destas aplicações são alvo de ataques constantemente por sua alta disponibilidade. Portanto, a introdução de vulnerabilidades deve ser evitada desde o início do desenvolvimento da aplicação. O *framework* Laravel fornece recursos suficientes para que uma aplicação Web seja desenvolvida sem nenhuma das vulnerabilidades abordadas neste trabalho, mas a responsabilidade de utilizar estes recursos corretamente é dos desenvolvedores.

REFERÊNCIAS

GASSER, M. **Building a Secure Computer System**. New York: Van Nostrand Reinhold Co., 1988.

BISHOP, M. **Introduction to Computer Security**. [S. l.]: Addison-Wesley, 2005.

KUROSE, J. F.; ROSS, K. W. **Computer Networking**: A Top-Down Approach. 6th ed. [S. l.]: Pearson, 2012.

GROFF, J.; WEINBERG, P. **SQL the Complete Reference**. 3rd ed. New York: McGraw-Hill, 2009.

VIEIRA, M.; ANTUNES, N.; MADEIRA H. Using web security scanners to detect vulnerabilities in web services. **2009 IEEE/IFIP International Conference on Dependable Systems & Networks**, Lisbon, p. 566-571, 2009.

KROMBHOLZ, K.; HOBEL, H.; HUBER, M.; WEIPPL, E. Advanced Social Engineering Attacks. **Journal of Information Security and Applications**, New York, v. 22, p. 113-122, jun. 2015.

IRANI, D.; BALDUZZI, M.; BALZAROTTI, D.; KIRDA, E.; PU, C. Reverse Social Engineering Attacks in Online Social Networks. **DIMVA'11 Proceedings of the 8th international conference on Detection of intrusions and malware, and vulnerability assessment**, Amsterdam, p. 55-74, jul. 2011.

JOSHI, J. B. D.; AREF, W. G.; GHAFOOR, A.; SPAFFORD, E. H. Security Models for Web-based Applications. **Communications of the ACM**, New York, v. 44, n. 2, p. 38-44, fev. 2001.

REIS, C.; BARTH, A.; PIZANO, C. Browser Security: Lessons from Google Chrome. **Communications of the ACM**, New York, v. 52, n. 8, p. 45-49, ago. 2009.

BISHOP, M.; BAILEY, D. A critical analysis of vulnerability taxonomies. **Technical Report CSE-96-11**, Department of Computer Science at the University of California at Davis, set. 1996.

FONSECA, J.; VIEIRA, M.; MADEIRA. H. Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks. **13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)**, Melbourne, p. 365-372, dez. 2007.

ANTUNES, N.; VIEIRA, M. Defending against Web Application Vulnerabilities. **Computer**, [S. l.], v. 45, n. 2, p. 66-72, fev. 2012.

KELLEY, P. G.; KOMANDURI, S.; MAZUREK, M. L.; SHAY, R.; VIDAS, T.; BAUER, L.; CHRISTIN, N.; CRANOR, L. F.; LÓPEZ, J. Guess Again (and Again and Again): Measuring Password Strength by Simulating Password-Cracking Algorithms, **2012 IEEE Symposium on Security and Privacy**, San Francisco, p. 523-537, mai. 2012.

SLAVIERO, M. TLS/SSL and .NET Framework 4.0. **Red Gate Simple Talk**, [S. l.], set. 2011. Disponível em: <https://www.red-gate.com/simple-talk/dotnet/net-framework/tlsssl-and-net-framework-4-0/#second>. Acesso em: 04 set. 2017.

W3Techs. Usage of web servers for websites. **Web Servers**, [S. l.], set. 2017. Disponível em: <https://w3techs.com/technologies/overview/web_server/all>. Acesso em: 06 set. 2017.

ModSecurity. What Can ModSecurity Do? **About**, [S. l.], 2017. Disponível em: <https://modsecurity.org/about.html>. Acesso em: 07 set. 2017.

OWASP. Os dez riscos de segurança mais críticos em aplicações web. **OWASP Top 10**, [S. l.], jun. 2013. Disponível em: <https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project>. Acesso em: 09 set. 2017.

WhiteHat Security. Most Serious Vulnerabilities By Class. **2017 Application Security Statistics Report**, [S. l.], v. 12, p. 15-16, 2017. Disponível em: <https://www.whitehatsec.com/resources-category/premium-content/web-application-stats-report-2017/>. Acesso em: 09 set. 2017.

CAMPANILE, C. DEM POL'S SON WAS 'HACKER'. **New York Post**, [S. l.], set. 2008. Disponível em: <http://nypost.com/2008/09/19/dem-pols-son-was-hacker/>. Acesso em: 16 set. 2017.

SEACORD, R. Input Validation and Data Sanitization. **SEI CERT Oracle Coding Standard for Java**, [S. l.], abr. 2015. Disponível em: <https://www.securecoding.cert.org/confluence/display/java/Input+Validation+and+Data+Sanitization>. Acesso em: 03 out. 2017.

HashiCorp. Introduction to Vagrant. **What is Vagrant?**, [S. l.]. 2017. Disponível em: <https://www.vagrantup.com/intro/index.html> Acesso em: 05 nov. 2017.

OTWELL, T. Introduction. **Laravel Homestead**, [S. l.], 2017. Disponível em: <https://laravel.com/docs/5.5/homestead#introduction>. Acesso em: 05 nov. 2017.

SCHNEIER, B. The Blowfish Encryption Algorithm. **Schneier on Security**. [S. l.], 2017. Disponível em: <https://www.schneier.com/academic/blowfish/>. Acesso em: 15 nov. 2017.

BLAZEJUK, A. Internet Forum. **GitHub**, [S. l.], 2017. Disponível em: <https://github.com/ablazejuk/internet_forum>. Acesso em: 10 dez. 2017.