INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA SUL-RIO-GRANDENSE - IFSUL, CAMPUS PASSO FUNDO
CURSO DE TECNOLOGIA EM SISTEMAS PARA INTERNET

VICTOR HUGO FOLCHINI SEBBEN

SEGURANÇA EM SISTEMAS WEB:
UMA ANÁLISE DA EXTENSÃO PDO COMO FORMA DE PROTEÇÃO DE SISTEMAS PHP CONTRA INJEÇÕES DE SQL

Wilian Bouviér

PASSO FUNDO, 2014

VICTOR HUGO FOLCHINI SEBBEN

SEGURANÇA EM SISTEMAS WEB: UMA ANÁLISE DA EXTENSÃO PDO COMO FORMA DE PROTEÇÃO DE SISTEMAS PHP CONTRA INJEÇÕES DE SQL

Monografia apresentada ao Curso de Tecnologia em Sistemas para Internet do Instituto Federal Sul-Rio-Grandense, *Campus* Passo Fundo, como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet.

Orientador: Wilian Bouviér

PASSO FUNDO, 2014

VICTOR HUGO FOLCHINI SEBBEN

SEGURANÇA EM SISTEMAS WEB: UMA ANÁLISE DA EXTENSÃO PDO COMO FORMA DE PROTEÇÃO DE SISTEMAS PHP CONTRA INJEÇÕES DE SQL

Trabalho de Conclusão de Curso aprovado em ____/____/____ como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet

Banca Examinadora:

____
Nome do Professor(a) Orientador(a)

____
Nome do Professor(a) Convidado(a)

____
Nome do Professor(a) Convidado(a)

____
Coordenação do Curso

PASSO FUNDO, 2014

DEDICATÓRIA

Dedico a meus pais, pelo amor incondicional e pelo exemplo de honestidade e sabedoria.
Dedico a minha irmã Bruna, exemplo de coragem.
Dedico a minha tia Dolores, por todo o carinho que sempre dispensou a mim e toda a família.
Dedico a minha noiva Daniele, exemplo de garra, pela compreensão e apoio.
Dedico a meu colega e amigo Fernando, pelo auxílio, conselhos e pelo companheirismo ao longo de toda essa trajetória.
Dedico a meu orientador Wilian, pela paciência, conselhos e ensinamentos.

AGRADECIMENTOS

Agradeço a meus pais, Tereza e José, cujo apoio foi essencial em todas as minhas conquistas. Agradeço seu carinho, compreensão e amor. Sem seus ensinamentos e sabedoria, sem sua ajuda, sem seu exemplo de coragem e determinação, eu nada seria.

Agradeço a minha noiva Daniele que, com seu exemplo, me ensina diariamente a enfrentar os desafios com coragem e determinação. Agradeço seu amor e compreensão.

A minha irmã Bruna, sempre presente para me auxiliar, aconselhar e apoiar. Sempre divertida e bem-humorada, sua presença torna minha vida mais alegre e leve.

A minha tia Dolores, que não poupa esforços para ajudar, e dá sempre exemplo de serenidade e sabedoria.

O amor de minha família pela leitura e pelo conhecimento converteram-se em incentivos para estudar, conhecer e saber mais. Agradeço imensamente por esse legado.

Agradeço também a meu amigo Fernando, programador, apaixonado por Linux e mestre Jedi. Seu espírito *hacker* (de quem se diverte em conhecer sempre mais) é fonte constante de inspiração.

Agradeço ao professor Alexandre Lazzaretti que, com suas considerações, auxiliou para que a presente pesquisa se tornasse melhor.

Agradeço ao meu professor orientador, Wilian Bouviér, pela dedicação, empenho, paciência e conselhos no longo processo de elaboração dessa pesquisa.

Por fim, agradeço ao IFSUL, seus professores e funcionários que, com seu trabalho, tornam possível a construção e disseminação do conhecimento, tarefa das mais nobres.

EPÍGRAFE

"Deveria ser um truísmo que a literatura científica existe para disseminar conhecimento científico, e que publicações científicas existem para facilitar tal processo. Segue, portanto, que regras para o uso da literatura científica deveriam ser projetadas para ajudar a atingir esse objetivo.

...

Eles usam a lei de copyright, que está ainda em vigor, apesar de sua impropriedade para as redes de computador, como desculpa para impedir que os cientistas escolham novas regras.

Pelo bem da cooperação científica e do futuro da humanidade, devemos rejeitar essa abordagem em sua raiz – não apenas os sistemas obstrutivos que foram instituídos, mas as prioridades errôneas que os inspiraram."

(Richard Stallman)

RESUMO

A presente pesquisa destina-se a explorar o tema da segurança de sistemas web, com ênfase particular às diferentes formas de ataques por injeção de SQL. Visa, ainda, averiguar se a PDO, API da linguagem PHP para conexão com banco de dados, é eficaz na proteção de sistemas contra essa forma de ataque, por meio do uso de *prepared statements*. A pesquisa encontra motivação na necessidade de proteger dados contra acessos indevidos, em especial por parte de usuários mal-intencionados, cujas ações podem causar grandes prejuízos a pessoas e organizações. Testes realizados em um sistema de *login* revelaram que o uso de *prepared statements* e a parametrização das informações entradas pelo usuário e utilizadas em consultas ao banco de dados são altamente eficazes na proteção de sistemas contra injeções de SQL.

Palavras-chave: injeção de SQL, segurança, sistemas web.

ABSTRACT

This paper aims to look into the subject of security in web-based systems, with an emphasis on the different forms of SQL injection attacks. It also aims to verify if PDO, a PHP API for dealing with database connection, is effective in preventing this kind of attack, by using prepared statements. This research finds its motivation in the need to protect data against undue access, specially by malicious users, whose actions can cause a great deal of trouble to people and organizations. Tests conducted in a login system showed that the use of prepared statements and the parametrization of information entered by the user and used in queries to the database are very effective in protecting systems against SQL injections.

Keywords: SQL injection, security, web systems.

LISTA DE FIGURAS

Figura 1: Informações da tabela "cidades"..21
Figura 2: Informações da tabela "estados"...22
Figura 3: O comando SELECT é utilizado para obter informações de tabelas........................22
Figura 4: Comando SELECT com uso da palavra-chave WHERE...22
Figura 5: Comando JOIN: usado para buscar dados de tabelas relacionadas..........................23
Figura 6: Comando OUTER JOIN selecionando informações de todos os estados, com ou sem cidades relacionadas..23
Figura 7: O comando UNION une resultados de duas ou mais consultas...............................24
Figura 8: Resultado da consulta usando UNION...24
Figura 9: Criação de uma função no MariaDB..25
Figura 10: Criação de um stored procedure no MariaDB..25
Figura 11: Injeção de tautologia: a comparação “1=1” resultará sempre verdadeira................33
Figura 12: Injeção que visa gerar erros no banco..34
Figura 13: Stored procedure que autentica um usuário..35
Figura 14: Primeira etapa de uma injeção cega...36
Figura 15: Segunda etapa de uma injeção cega...37
Figura 16: Injeção baseada em tempo, fazendo uso da instrução WAITFOR..........................37
Figura 17: Injeção baseada em codificações alternativas..38
Figura 18: Exemplo de uso da ferramenta SQLMap, passando uma URL como parâmetro....41
Figura 19: Página com formulário de login...43
Figura 20: Método selectUser implementado sem prepared statements: uma string é montada com o comando SELECT e os dados entrados pelo usuário...44
Figura 21: Prepared statement criado a partir de um objeto PDO. A partir disso, o comando preparado pode ser executado com diferentes parâmetros...44
Figura 22: Métodos selectUser e login, na classe User, são responsáveis pela autenticação do usuário no sistema. O método selectUser aparece, aqui, implementado com a utilização de prepared statements e parametrização...45
Figura 23: Injeção inserida no campo de usuário, no formulário de login..............................46
Figura 24: Resultado mostrado na tela: o campo de usuário mostrou-se vulnerável à injeção 47
Figura 25: Resposta dada pelo sistema à injeção piggy-backed. Apesar da mensagem adequada mostrada na tela, a injeção obteve sucesso...47
Figura 26: A execução da ferramenta SQLMap encontrou vulnerabilidades no parâmetro

username...48
Figura 27: Resultado da primeira execução da SQLMap...49
Figura 28: Informações mostradas no segundo teste executado com a SQLMap.....................49

LISTA DE ABREVIATURAS E SIGLAS

OWASP – Open Application Security Project, p. 11.

PHP – PHP: Hypertext Preprocessor, p. 11.

PDO – PHP Data Objects, p. 11.

SQL – Structured Query Language, p. 11.

URI – Unified Resource Identifier, p. 14.

HTML – Hypertext Markup Language, p. 15.

CSS – Cascading Style Sheets, p. 15.

SGBD – Sistema de Gerência de Banco de Dados, p. 18.

DML – Data Manipulation Language, p. 21.

DDL – Data Definition Language, p. 21.

DCL – Data Control Language, p. 21.

API – Application Programming Interface, p. 27.

PDO – PHP Data Objects, p. 27.

ASCII – American Standard Code for Information Interchange. p. 37.

SUMÁRIO

1 INTRODUÇÃO....................11
1.1 MOTIVAÇÃO....................12
1.2 OBJETIVOS....................13
1.2.1 Objetivo geral....................13
1.2.2 Objetivos específicos....................13
2 REFERENCIAL TEÓRICO....................14
2.1 AS PÁGINAS WEB: UM BREVE HISTÓRICO....................14
2.1.1 Do que são feitas as páginas web: as tecnologias envolvidas....................15
3 CRESCIMENTO DAS APLICAÇÕES WEB: SISTEMAS POR TRÁS DAS PÁGINAS...17
3.1 PHP....................17
3.2 BANCOS DE DADOS....................18
3.2.1 Modelo relacional....................19
3.2.2 A linguagem SQL....................21
3.2.3 Principais sistemas de gerência de bancos de dados....................26
3.3 APIS PARA CONEXÃO COM BANCO DE DADOS E PDO....................27
4 SEGURANÇA EM SISTEMAS WEB....................30
4.1 ATAQUES POR INJEÇÃO DE CÓDIGO....................31
4.2 ATAQUES POR INJEÇÃO DE SQL....................31
4.2.1 Formas de ataques por injeção de SQL....................32
4.2.2 Prevenção de ataques por injeção de SQL....................38
4.3 TESTES....................39
4.3.1 Ferramentas para testes e SQLMap....................40
5 IMPLEMENTAÇÃO E EXECUÇÃO DE TESTES....................42
5.1 IMPLEMENTAÇÃO DO SISTEMA DE LOGIN....................42
5.2 REALIZAÇÃO DE TESTES DE PENETRAÇÃO....................45
5.2.1 Injeções realizadas em sistema sem prepared statements....................46
5.2.2 Injeções realizadas em sistema parametrizado com prepared statements....................48
6 CONSIDERAÇÕES FINAIS....................51
REFERÊNCIAS....................53

1 INTRODUÇÃO

A Internet possibilitou o compartilhamento de informações dos mais diversos tipos, acelerando a comunicação e oferecendo as respostas rápidas de que a sociedade moderna precisa. Tendo em vista que, através da Internet, são transmitidas informações relevantes para pessoas e organizações, e cujo acesso precisa ser restringido, a segurança é sempre um tema pertinente, tanto do ponto de vista das redes e serviços de rede como da programação de sistemas.

No que diz respeito à segurança, aplicações web apresentam, em relação a outros sistemas, um agravante: o fato de que o número de usuários pode ser muito grande, e ataques podem vir de maneira mais ou menos anônima. Vulnerabilidades, definidas pela OWASP[1] como “falhas ou fraquezas no design, implementação, operação ou gerenciamento que poderiam ser exploradas para comprometer os objetivos de segurança do sistema” (MEUCCI e MULLER, p. 27), precisam ser detectadas e consertadas, para que as informações estejam acessíveis somente a quem sobre elas tiver direito.

PHP[2] é uma linguagem utilizada especialmente para a confecção de sistemas web. A presente pesquisa explora conceitos relacionados à segurança de páginas web e sistemas PHP, procurando averiguar até que ponto a utilização da extensão PDO[3], fornecida pelos próprios desenvolvedores da linguagem, pode tornar um sistema mais seguro, coibindo diversas formas de ataque por injeção de SQL[4] às quais ele poderia, de outra forma, estar vulnerável.

A fim de prevenir essa forma de ataque, é necessário validar os dados inseridos pelos usuários (ou seja, checar as informações inseridas antes de usá-las na aplicação). A extensão PDO é capaz de prover uma camada de proteção por meio do uso de *prepared statements*, oferecendo uma implementação, em tese, segura contra injeções de SQL. Daí advém a importância de conhecer essa tecnologia melhor.

O capítulo 2 do trabalho apresenta um referencial teórico que embasará o leitor a respeito das tecnologias envolvidas na construção de sistemas web. O capítulo 3 aborda de maneira mais específica a linguagem de programação PHP, bem como a API para conexão com banco de dados PDO, e a tecnologia de bancos de dados relacionais (incluindo a linguagem SQL). Em seguida, aborda o tema da segurança, explorando e exemplificando as formas de ataques por injeção de SQL. O capítulo 4, por sua vez, documenta a implementação

---

1 OWASP: *Open Application Security Project*, ou Projeto Aberto de Segurança de Aplicações Web.
2 PHP: acrônimo recursivo para *PHP: Hypertext Preprocessor*.
3 PDO: *PHP Data Objects* (objetos de dados da PHP).
4 SQL: *Structured Query Language* (linguagem de consulta estruturada).

de um sistema de *login* simples, desenvolvido a partir das tecnologias PHP, PDO e MariaDB, e os testes de penetração realizados sobre ele para averiguar o nível de proteção oferecido pela PDO.

A presente pesquisa objetivou converter-se em uma fonte de informações útil, no contexto da programação para Internet com PHP, abordando uma temática cuja importância é crescente, e a respeito da qual é necessário construir conhecimento. Dessa forma, espera-se que o trabalho resultante desse esforço possa trazer algum benefício a desenvolvedores e instituições, bem como ao meio acadêmico.

1.1 MOTIVAÇÃO

A questão da segurança permeia praticamente todas as atividades relacionadas à tecnologia digital. Informações representam ativos para empresas, organizações e pessoas e, por isso, muitas vezes, necessitam ter seu acesso restringido.

Nesse cenário, é importante trabalhar para garantir que os níveis de segurança dos sistemas estejam adequados. Quanto menor a preocupação com segurança – e, consequentemente, o esforço em promovê-la –, maior tende a ser a vulnerabilidade da proteção ao acesso de informações relevantes.

Conforme ressaltam Snyder, Myer e Southwell, embora seja difícil crer que alguém trataria informações valiosas de maneira descuidada, facilitando o acesso indevido a elas, isso ocorre com grande frequência no mundo computacional. Os autores afirmam que, apesar da retórica ser sempre a de que segurança é importante, em muitas ocasiões ela é deixada de lado, como um assunto secundário (2010, p. 1, 9). Esse descuido pode advir, é claro, da negligência, mas frequentemente vem do desconhecimento.

Uma das motivações para a realização desta pesquisa foi o desejo de conhecer mais a respeito desse assunto, em particular no que diz respeito a formas de ataque por injeção de SQL (uma preocupação frequentemente presente em sistemas web) e maneiras de evitá-las.

Conforme Halfond, Viegas e Orso, aplicações web vulneráveis a ataques por injeção de SQL podem acabar por permitir completo acesso de usuários mal-intencionados aos bancos de dados e, consequentemente, a todo tipo de informação neles armazenada. Em estudo citado pelos autores, promovido pelo Gartner Group, no qual 300 sites web foram analisados, a grande maioria das páginas revelaram-se vulneráveis a esse tipo de ataque (2006).

A linguagem PHP é utilizada para a confecção de sistemas web. A extensão PDO é fornecida oficialmente pela comunidade que desenvolve a linguagem, e seu uso é

recomendado na prevenção contra injeções de SQL, em função de seu suporte a *prepared statements*. Portanto, é importante chamar a atenção da comunidade acadêmica e empresas para o uso dessa ferramenta, bem como, mais genericamente, para a necessidade de ações voltadas à segurança de páginas e sistemas.

Assim, a realização de uma pesquisa voltada ao estudo de ataques por injeção de SQL e formas de evitá-los encontrou sua principal motivação na possibilidade de trazer à tona a importância de pensar a respeito de segurança e, na medida do possível, explorar algumas técnicas que podem ser utilizadas para o desenvolvimento de sistemas menos vulneráveis a ações perniciosas.

1.2 OBJETIVOS

1.2.1 Objetivo geral

Realizar um estudo de segurança de páginas e sistemas web, visando compreender melhor as formas de ataque por injeção de SQL e como evitá-las. A pesquisa se dará por meio de uma revisão bibliográfica seguida de um estudo que procurará, por meio de testes, averiguar o nível de segurança que a API PDO apresenta contra tais ataques.

1.2.2 Objetivos específicos

O projeto tem os seguintes objetivos específicos:

- compreender conceitos relacionados a páginas web e sistemas PHP;
- compreender conceitos relacionados à segurança de páginas e sistemas web, com ênfase para o estudo de ataques por injeção de SQL;
- investigar os principais tipos de injeções de SQL utilizadas no ataque a páginas web;
- desenvolver um sistema de login em PHP, fazendo uso da API PDO para conexão com banco de dados, visando averiguar sua eficácia na prevenção de ataques por injeção de SQL.

2 REFERENCIAL TEÓRICO

Este capítulo traz um embasamento teórico a respeito de páginas e sistemas web, abordando brevemente as tecnologias básicas envolvidas em sua construção.

2.1 AS PÁGINAS WEB: UM BREVE HISTÓRICO

Páginas web rapidamente tornaram-se parte do cotidiano. Seja para obter informações acerca de, por exemplo, o horário de um ônibus, ou para realizar pesquisas mais elaboradas, como um trabalho de aula, buscas são frequentemente realizadas na Internet, e acabam levando a uma ou mais páginas que oferecem as informações necessárias.

A *World Wide Web* foi inventada no ano de 1989, por Tim Berners-Lee, cerca de 20 anos após o surgimento de redes que, posteriormente, dariam origem ao que hoje chama-se Internet (HISTORY, 2014). Tal criação representou um passo crucial para que a Internet se tornasse a tecnologia popular que é hoje (WILLIAMS; TOLLETT, 2001, p. 9).

Pode-se definir a *World Wide Web*, ou simplesmente web, como “um espaço de informações no qual os itens de interesse, denominados recursos, são identificados por identificadores globais chamados *Uniform Resource Identifiers* (URI)[1]” (ARCHITECTURE, 2004, tradução nossa). Portanto, enquanto a Internet se refere a toda a estrutura que permite a comunicação de máquinas por meio de redes, a web ocupa-se em definir como o conteúdo que trafega através dessas redes será apresentado e localizado.

A primeira página web foi criada e tornada acessível na rede no ano de 1990, por iniciativa do próprio Berners-Lee (HISTORY, 2014). Cabe, portanto, a questão: o que são páginas web? Para Williams e Tollett, elas são, basicamente, “páginas de texto com mensagens codificadas dizendo ao *browser*[2] o que fazer.” Em outras palavras, são documentos criados para serem lidos, através da Internet, por meio de navegadores (2001, p. 38).

O W3 Consortium, instituição responsável por definir as especificações de tecnologias envolvidas na web, define uma página web como o resultado do processamento realizado quando o usuário, por meio do navegador, requisita determinado conteúdo *online*, recebendo as informações que são, então, interpretadas e mostradas em seu navegador (o que é visível, então, ao usuário, é a página propriamente dita) (ARCHITECTURE, 2004).

---

1 *Unified Resource Identifier*, ou URI, refere-se à tecnologia de endereçamento da Web. É uma cadeia de caracteres (string) usada para identificar (dar um nome) a recursos na web (NAMING, 2014).

2 O *browser*, ou navegador (em português), é um software que permite ao usuário acessar páginas web através da Internet.

Quando se acessa web sites, frequentemente interage-se não somente com páginas isoladas, mas também com aplicações web. Uma aplicação web pode ser definida como um programa que executa determinado tipo de função, geralmente acessado através de um navegador (20 THINGS, 2010, p. 10).

Uma aplicação web é geralmente composta por três componentes principais: um navegador, um servidor de aplicação e um servidor de banco de dados. Ela atua no sentido de buscar dados no banco e apresentá-los ao usuário, frequentemente com base nas informações entradas, previamente, pelo próprio usuário (SU e WASSERMANN, 2006). Vale ressaltar que, em especial na tecnologia móvel (celulares e *tablets*), muitas vezes essas aplicações são executadas por meio de interfaces próprias, sem o uso do navegador.

2.1.1 Do que são feitas as páginas web: as tecnologias envolvidas

A principal linguagem utilizada na confecção de páginas web é chamada HTML, acrônimo que, em inglês, advém dos termos *hypertext markup language*, ou linguagem de marcação de hipertexto. Segundo as especificações oficiais da HTML 5, acessíveis através do site do World Wide Web Consortium, a HTML é uma linguagem de marcação que visa definir, semanticamente, a estrutura de documentos. Isso significa que sua função é indicar o que cada uma das partes de um documento representa. Seu uso não é mais recomendado para definir a apresentação (aparência) de uma página (HTML5, 2014).

Para definir a apresentação de páginas web, ou seja, sua aparência, a W3C desenvolveu as especificações da CSS (*cascading style sheets*, ou folhas de estilo em cascata). Enquanto a HTML define páginas semanticamente (em termos de significado, ou o que cada parte representa), a CSS pode ser usada para selecionar elementos constituintes de documentos HTML e modificar sua aparência (WYKE-SMITH, 2013, p. 1).

Outra tecnologia importante no desenvolvimento da web é a linguagem JavaScript. Os códigos programados nessa linguagem rodam no navegador, permitindo que as páginas se tornem interativas, isto é, fazendo com que elas ofereçam respostas às ações do usuário (MORRISON, 2008, p. 4).

Tais tecnologias permitiram que as páginas web se tornassem cada vez mais interessantes e complexas. No entanto, há muito tempo elas deixaram de ser apenas documentos para serem lidos e/ou vistos através da Internet. As aplicações web passaram a oferecer mais e mais recursos e, hoje, sistemas são construídos atrás dessas páginas. E mesmo documentos *online* aparentemente simples são, inúmeras vezes, gerados dinamicamente, ou

seja, pelo próprio programa de computador acessado através do navegador. Na próxima seção, aborda-se os sistemas web e algumas das tecnologias envolvidas em seu desenvolvimento.

3 CRESCIMENTO DAS APLICAÇÕES WEB: SISTEMAS POR TRÁS DAS PÁGINAS

Com a disseminação da Internet e, particularmente, da web, os programas de computador, antes executados, na maioria das vezes, em máquinas isoladas, passaram a permitir que computadores de todo o mundo se comuniquem (DEITEL; DEITEL, 2010, p. 5).

A partir daí, diversas linguagens, algumas criadas inicialmente para outros propósitos, passaram a ser usadas para o desenvolvimento de sistemas web. A PHP é utilizada para esse propósito (para o qual, aliás, nasceu). Por se tratar do real foco do presente trabalho, serão abordadas um pouco de sua história e características a seguir.

Outra tecnologia relevante nesse contexto é o banco de dados, assunto que será tratado mais adiante, com ênfase para bancos de dados relacionais.

3.1 PHP

A tecnologia PHP surgiu não muito depois da própria web. Em 1995, Rasmus Lerdorf desenvolveu um *script*, utilizando a linguagem Perl, que permitia registrar informações de visitas ao seu currículo *online*. Lerdorf decidiu distribuir seu conjunto de ferramentas gratuitamente para quem quisesse dele fazer uso, e chamou-o Personal Home Page, ou PHP (o significado das letras foi alterado, e hoje indicam o acrônimo recursivo PHP: *Hypertext Preprocessor*, ou PHP: Preprocessador de Hipertexto) (GILMORE, 2010, p. 2).

Como, nos primórdios da web, tais ferramentas eram muito raras, ou mesmo inexistentes, a PHP se tornou rapidamente bastante popular. Incentivado pelo sucesso de sua criação, Lerdorf deu continuidade ao desenvolvimento, passando a utilizar C em lugar de Perl para programar o núcleo dessa nova linguagem de programação. Em 1997, a versão 2.0 seria lançada. Em 1998, com o lançamento da versão 3.0 (já com novos desenvolvedores auxiliando Lerdorf), a PHP contava com mais de 50.000 usuários. Pouco depois, em 1999, a Netcraft, empresa de análise e pesquisa sobre a Internet, estimou que mais de um milhão de pessoas já estavam usando a linguagem em suas páginas web (GILMORE, 2010, p. 2).

Dois desenvolvedores da linguagem, Zeev Suraski e Andi Gutmans, foram responsáveis por reescrever o interpretador PHP, modificando substancialmente o funcionamento da linguagem. A partir desse esforço, surgiria a PHP 4, em 2000. Primeira grande versão, acrescentava suporte nativo a sessões e algumas funcionalidades – embora ainda rudimentares – de orientação a objetos, entre muitos outros novos recursos (GILMORE, 2010, p. 2 e 3).

A versão 5 trouxe mais melhorias. Uma das mais significativas foi o suporte avançado à orientação a objetos (GILMORE, 2010, p. 4). No momento de escrita desse trabalho, a última versão estável é a 5.6.2.

PHP é uma linguagem que recebeu, e ainda recebe, inúmeras críticas. Jeff Atwood, programador e escritor cujo trabalho pode ser lido no blog Coding Horror, ressalta a falta de organização na linguagem, com inúmeras funções predefinidas com pouca padronização nos nomes[1]. No entanto, sua importância justifica o interesse acadêmico em pesquisar sobre ela. Afinal, como o próprio autor citado ressalta, alguns dos sites mais importantes atualmente foram feitos com PHP: Facebook, Wikipedia, YouTube, entre muitos outros.

Além disso, como ressalta o desenvolvedor Coen Jacobs, em artigo publicado em seu site, muitos dos erros que comumente são apontados na linguagem dizem respeito a versões antigas, ou mesmo a más práticas de programação[2].

Um dos avanços importantes que a linguagem trouxe, mais recentemente, visando ampliar a segurança e facilitar a interação com bancos de dados, foi a extensão PDO, assunto relevante para a pesquisa, e que será tratado mais adiante.

3.2 BANCOS DE DADOS

Um banco de dados pode ser definido como "uma coleção de dados estruturados, organizados e armazenados de forma persistente por uma aplicação informatizada". O termo "persistente" significa que os dados serão armazenados em memória permanente (ou seja, no disco rígido) (DAMAS, 2007, p. 31).

Tais dados precisam ser gerenciados de maneira adequada. A gerência de bancos de dados é realizada por aplicações específicas chamadas sistemas de gerência de banco de dados (SGBD). Um SGBD "é uma aplicação informatizada que fornece a *interface* entre os dados que são armazenados fisicamente no *banco de dados* e o usuário (pessoa ou aplicação) desses dados" (DAMAS, 2007, p. 32, grifo do autor).

Portanto, em um sistema, a aplicação feita pelo desenvolvedor irá interagir com um SGBD para fazer uso de um banco de dados, de modo a armazenar informações permanentemente e acessá-las mais tarde.

Date destaca 7 funções que um SGBD precisa desempenhar. A primeira delas, a definição de dados, diz respeito à capacidade de "aceitar definições de dados […] em formato

1 O artigo está disponível em http://blog.codinghorror.com/php-sucks-but-it-doesnt-matter/.
2 O artigo está disponível em http://coenjacobs.me/php-is-not-bad/

fonte e convertê-los para o formato objeto apropriado" (o SGBD deve ser capaz de compreender comandos de definição de dados – como a criação de tabelas – e analisar e responder a pedidos de manipulação dos dados). A segunda função é a manipulação de dados ("buscar, atualizar ou excluir" informações existentes ou "acrescentar novos dados"). Outra função é a otimização e execução (antes de implementar uma requisição, o SGBD deve planejar uma maneira otimizada de fazê-lo). Em seguida, tem-se a segurança e integridade de dados, que diz respeito à ação de rejeitar tentativas de violar restrições de segurança configuradas no sistema. Outra função importante é a recuperação de dados e concorrência (controle de recuperação de dados e de concorrência no acesso a eles). Há, ainda, as funções de dicionário de dados (é como um "banco de dados do sistema", contendo metadados, ou seja, informações sobre as informações armazenadas no banco) e, por fim, desempenho (o SGBD deve ser tão eficiente quanto possível no desempenho de suas funções) (2003, p. 38-40).

3.2.1 Modelo relacional

Antes do surgimento de SGBDs, informações eram armazenadas em arquivos de texto. Tal abordagem, no entanto, apresentava sérias limitações. Se aplicações distintas precisassem manter registro dos mesmos dados, teriam de ter arquivos próprios (cópias dos dados), ocasionando redundância e dificuldade de manutenção. Se pudessem acessar um mesmo arquivo, aplicações diferentes poderiam ter processos de validações de dados diferentes, dificultando a garantia de integridade dos dados. Além disso, precisavam ser programadas especificamente para lidar com determinado formato de arquivo, e não era possível estabelecer relações entre arquivos (era necessário implementá-las no código do programa). Preocupações com a segurança também existiam, já que não é possível restringir o acesso a parte dos dados em um arquivo para determinado usuário (DAMAS, 2007, p. 39 e 40).

O sistema de gerência por arquivos evoluiu, então, para o uso de SGBDs. Uma das primeiras tentativas foi o modelo hierárquico, no qual as informações ficavam armazenadas em árvore, estabelecendo uma relação pai-filho entre os dados. Os registros eram "ligados hierarquicamente entre si através de *links"* (DAMAS, 2007, p. 41, grifo do autor).

O banco de dados hierárquico apresentava bom desempenho no acesso sequencial dos dados. No entanto, a estrutura em árvore dificultava o acesso aleatório, e era ineficiente em relações do tipo N:N (muitos para muitos), caso em que provocava replicação dos dados na estrutura (DAMAS, 2007, p. 44).

Como alternativa, surgiu, então, o modelo em rede. Os dados passaram a ser representados na forma de uma rede de registros ligados entre si por meio de links. Eliminando a hierarquia, esse modelo permitia que um mesmo registro possuísse várias associações, em lugar de relacionar-se somente com o nodo pai, como no modelo anterior. No entanto, também apresentava limitações, em especial em relação à dificuldade de implementação (DAMAS, 2007, p. 44 e 45).

Devido a seus problemas inerentes, os modelos hierárquico e em rede acabaram fadados ao obsoletismo. Em 1970, surgiu, então, o modelo relacional, com base na teoria matemática dos conjuntos. Apesar de restrito, inicialmente, ao meio acadêmico, por limitações de tecnologia de hardware da época, acabou por tornar-se massivamente usado, popularizando a própria utilização da tecnologia de bancos de dados (DAMAS, 2007, p. 46 e 47).

No modelo relacional, os dados são armazenados em tabelas ou relações compostas por colunas (atributos, que são associados a um determinado tipo de dado) e linhas (registros ou tuplas) (DAMAS, 2007, p. 47).

As relações entre dados, em lugar de serem estabelecidas por meio de links, passaram a ser feitas através de colunas comuns em tabelas distintas. Cada linha de uma tabela é identificada por uma coluna ou combinação de colunas denominada “chave primária”. O valor ou combinação de valores deve ser único para cada linha. Para estabelecer a relação entre duas tabelas, a chave primária de uma torna-se um campo – denominado “chave estrangeira” – na outra (HEUSER, 2009).

O modelo relacional trouxe inúmeros benefícios: podia-se “fazer alterações no esquema do banco de dados sem afetar a capacidade do sistema em manipular os dados” (DAMAS, 2007, p. 50).

Outros modelos de bancos de dados surgiram depois do relacional. Em particular, o modelo orientado a objetos tende a crescer de agora em diante, uma vez que o conceito de orientação a objetos assumiu papel muito importante na programação de sistemas. Em lugar de utilizar uma tabela, esse novo paradigma passa a permitir que objetos sejam diretamente armazenados no banco (DAMAS, 2007, p. 53).

Visando facilitar a interação de sistemas orientados a objetos com bancos de dados sem, no entanto, perder as funcionalidades de bancos relacionais, diversos SGBDs, hoje, caracterizam-se como objeto-relacionais (DAMAS, 2007, p. 55).

A presente pesquisa procurou aprofundar-se na segurança de sistemas web com base no modelo relacional, em particular no que diz respeito a injeções de SQL. A seguir, é abordada a linguagem SQL, componente essencial dos SGBDs relacionais.

3.2.2 A linguagem SQL

A linguagem SQL foi proposta juntamente com o modelo relacional de banco de dados, no ano de 1970, por Edgar Frank Codd. Ainda é considerada o padrão para esse tipo de sistema de gerência de banco de dados, sendo utilizada, com algumas diferenças, por todos eles, facilitando a vida de desenvolvedores e evitando “que se opte por arquiteturas proprietárias que implicam maiores custos de desenvolvimento” (DAMAS, 2007, p. 1 e 3).

A SQL não é uma linguagem multipropósito: sua função é permitir a manipulação de dados. É caracterizada como “ linguagem declarativa”, divergindo do caráter “procedimental” das linguagens tradicionais. Isso quer dizer que sua preocupação maior é descrever o que deverá ser feito (nesse caso, que dados buscar ou que operações realizar sobre eles), e não como deverá ser feito (DAMAS, 2007, p. 131).

Damas traz algumas funções importantes da linguagem SQL: “Criar, alterar e remover” elementos de um banco de dados (tabelas, *views*, etc.), “Inserir, alterar e apagar dados”, fazer consultas ao banco, controlar o acesso dos usuários aos dados e operações e garantir a “consistência e integridade dos dados” (2007, p. 3).

Para executar tais funções, a SQL apresenta três aspectos, referidos por Damas como suas sublinguagens: DML (*Data Manipulation Language,* ou linguagem de manipulação de dados), que inclui comandos para consulta e manipulação das informações contidas no banco (SELECT, INSERT, UPDATE, DELETE, entre outros); DDL (*Data Definition Language*, ou linguagem de definição de dados), cujos comandos permitem, entre outras operações, criar, alterar e remover tabelas (CREATE, ALTER e DROP, respectivamente); e DCL (*Data Control Language*, ou linguagem de controle de dados), que lida com permissões de usuários para executar determinadas operações (os principais comandos são GRANT e REVOKE, para atribuir e tirar permissão, respectivamente) (DAMAS, 2007, p. 131).

A seguir, explorar-se-á, um pouco mais a fundo, alguns dos comandos citados. Para os exemplos descritos abaixo, serão utilizadas as tabelas “cidades” e “estados”, contendo a estrutura e os dados mostrados nas figuras 1 e 2, respectivamente.

Figura 1: Informações da tabela "cidades"

```
+----+---------------+--------+
| id | nome          | estado |
+----+---------------+--------+
|  1 | Porto Alegre  |      1 |
|  2 | Passo Fundo   |      1 |
|  3 | Florianópolis |      2 |
+----+---------------+--------+
```

Fonte: Do autor

Figura 2: Informações da tabela "estados"

```
+----+-------+
| id | sigla |
+----+-------+
|  1 | RS    |
|  2 | SC    |
|  3 | SP    |
|  4 | MG    |
+----+-------+
```

Fonte: Do autor

Algumas observações importantes: a chave primária das tabelas está em seus respectivos campos “id” (que armazenam um número de identificação para cada linha). O campo “estado” da tabela “cidades” deverá conter um id existente na tabela “estados” (é uma chave estrangeira), estabelecendo a ligação entre as tabelas.

A SQL é uma linguagem relativamente simples. O comando mais comum é provavelmente o “SELECT”, cuja função é obter dados armazenados nas tabelas do banco. Para obter todos os dados da tabela cidades, por exemplo, pode-se utilizar a sintaxe descrita na figura 3.

Figura 3: O comando SELECT é utilizado para obter informações de tabelas

```
SELECT * FROM cidades;
```

Fonte: Do autor

O símbolo “*” indica que se quer obter as informações de todas as colunas da tabela (nesse caso, “id”, “nome” e “estado”). O resultado da consulta acima seria o que se encontra representado na figura 1, acima.

Pode-se utilizar a palavra-chave “WHERE” para executar uma operação de seleção, especificando uma condição para o retorno dos dados (CHURCHER, 2007, p. 173). Para exemplificar, pode-se selecionar o nome da cidade que tem o id igual a 1, conforme a figura 4.

Figura 4: Comando SELECT com uso da palavra-chave WHERE

```
SELECT nome FROM cidades WHERE id = 1;
```

Fonte: Do autor

Por vezes, é necessário obter dados de mais de uma tabela. Nesse caso, o comando “JOIN” pode ser usado (CHURCHER, 2007, P. 177). Se, por exemplo, queremos obter os

nomes das cidades acompanhados das siglas de seus respectivos estados, podemos proceder conforme indicado na figura 5.

Figura 5: Comando JOIN: usado para buscar dados de tabelas relacionadas

```
SELECT cidades.nome, estados.sigla
  FROM cidades
  INNER JOIN estados
    ON cidades.estado = estados.id;
```

Fonte: Do autor

Conforme ressalta Damas, o comando "INNER JOIN" (pode-se omitir o "INNER") é uma junção do tipo mais comum, ou seja, em que se reúnem informações de duas ou mais tabelas, "ligando-as através da *Chave Primária* de uma e da *Chave Estrangeira* da outra" (2007, p. 173, grifo do autor).

O comando "INNER JOIN" tem uma característica que, por vezes, pode não ser o comportamento desejado na consulta. Ele só retornará os dados da tabela "estados", por exemplo, que tiverem correspondência na tabela de "cidades" (só mostrará estados para os quais há cidades cadastradas). Pode-se obter uma relação completa das tabelas ligadas pela junção, mesmo quando não há correspondência na outra tabela, através do comando "OUTER JOIN" (DAMAS, 2007, p. 179). A figura 6 exemplifica seu uso.

Figura 6: Comando OUTER JOIN selecionando informações de todos os estados, com ou sem cidades relacionadas

```
SELECT cidades.nome AS cidade, estados.sigla AS estado
  FROM cidades
    RIGHT JOIN estados
      ON cidades.estado = estados.id;
```

Fonte: Do autor

Algumas considerações importantes: no "OUTER JOIN", pode-se mostrar todo o conteúdo da tabela à esquerda ou à direita. Neste caso, utilizou-se "RIGHT JOIN", já que quer-se mostrar todos os estados, e a tabela estados está à direita da cláusula JOIN (no comando "cidades RIGHT JOIN estados"). O comando AS apenas indica o nome pelo qual queremos que o campo seja chamado na tabela que mostra o resultado da consulta.

Por vezes, pode-se querer reunir o conteúdo de duas ou mais consultas ao banco de dados. A SQL torna isso possível por meio do comando UNION (DAMAS, 2007, p. 183). É possível selecionar, por exemplo, o id e o nome das cidades e o id e a sigla dos estados. Os resultados são mostrados em uma mesma tabela, e as colunas são denominadas de acordo com

a primeira consulta realizada. O código e a tabela de resultados são mostrados, respectivamente, nas figuras 7 e 8.

Figura 7: O comando UNION une resultados de duas ou mais consultas

```
SELECT cidades.id, cidades.nome FROM cidades
UNION
SELECT estados.id, estados.sigla FROM estados;
```

Fonte: Do autor

Figura 8: Resultado da consulta usando UNION

```
+----+----------------+
| id | nome           |
+----+----------------+
|  1 | Porto Alegre   |
|  2 | Passo Fundo    |
|  3 | Florianópolis  |
|  1 | RS             |
|  2 | SC             |
|  3 | SP             |
|  4 | MG             |
+----+----------------+
```

Fonte: Do autor

Damas ressalta uma restrição importante: "o número de campos a serem selecionados em cada um dos comandos SELECT tem de ser igual". Como será abordado posteriormente, o comando UNION pode ser usado em ataques bastante perigosos, uma vez que torna possível, em alguns casos, consultar mais informações do que uma aplicação naturalmente mostraria.

Os SGBDs relacionais oferecem, ainda, uma outra funcionalidade que pode se revelar útil: implementar parte da lógica do sistema no próprio SGBD (em lugar de programá-la na aplicação). Para tanto, é possível programar funções no banco, que funcionam de maneira análoga a funções na linguagem de programação (podem receber argumentos de entrada, processam dados e podem retornar valores) (CREATE, 2014).

Como exemplo, pode-se criar uma função que, recebendo o número de identificação de uma cidade, procura e retorna a sigla de seu estado. A declaração da função, em SQL (para o SGBD MariaDB), ficaria conforme a figura 9.

Figura 9: Criação de uma função no MariaDB

```
delimiter //
CREATE FUNCTION retorna_estado (id_cidade INT)
RETURNS CHAR(2) DETERMINISTIC
BEGIN
  DECLARE v_sigla CHAR(2);
  SELECT estados.sigla
    FROM estados
      JOIN cidades
        ON cidades.estado = estados.id
    WHERE cidades.id = id_cidade
    INTO v_sigla; RETURN v_sigla; END//
delimiter ;
```

Fonte: Do autor

Note-se que é preciso utilizar o comando *delimiter* antes e depois da função. Normalmente, o ';' delimita o fim de um comando no MariaDB. Como tal caractere será usado no corpo da função, é necessário trocar o delimitador de fim de comando para algo diferente. É comum usar-se, nesse caso, duas barras (//).

SGBDs são capazes de armazenar rotinas de código neles próprios, de modo que aplicações que os utilizem possam apenas chamar tais rotinas (sem a necessidade de implementá-las). Tal procedimento pode ser útil quando diferentes aplicações fazem uso do mesmo banco de dados, e necessitam realizar sobre ele operações iguais. O recurso de SGBDs que permite tais operações é chamado "stored procedure" (basicamente, o termo refere-se ao fato de que um procedimento – uma rotina de código – fica armazenada no SGBD, e pode ser chamada quando necessário) (STORED, 2014).

Como exemplo, pode-se criar um procedimento que receba, como parâmetros, o nome de uma cidade e o código de um estado, inserindo uma nova entrada na tabela "cidades", com a chave estrangeira do estado, como ilustrado na figura 10.

Figura 10: Criação de um *stored procedure* no MariaDB

```
delimiter //
CREATE PROCEDURE Inserir_cidade (nome_cidade VARCHAR(100), id_estado INT)
MODIFIES SQL DATA
BEGIN
  INSERT INTO cidades (nome, estado)
    VALUES (nome_cidade, id_estado);
END;//
delimiter ;
```

Fonte: Do autor

A SQL é, enfim, uma linguagem própria para a manipulação de informações. Além de consultar dados, ela também permite adicioná-los e removê-los, bem como fazer atualizações

neles. No entanto, os procedimentos de consulta são mais relevantes no contexto das injeções de SQL e, portanto, da presente pesquisa.

3.2.3 Principais sistemas de gerência de bancos de dados

Para aprofundar um pouco o conceito de SGBD, pode-se dizer que:

> Durante muitos anos a gerência dos dados foi realizada por aplicações informatizadas escritas, especificamente, para cada uma das situações que pretendia resolver. No entanto, com a evolução da tecnologia foram desenvolvidas aplicações específicas para a gerência de bancos de dados, independentemente do seu conteúdo ou objetivo (DAMAS, 2007, p. 32).

Tais aplicações são chamadas, genericamente, de sistemas de gerência de banco de dados. Nesse contexto, um programa desenvolvido, por exemplo, para a web, que deve armazenar e buscar dados em um banco, interage diretamente com o SGBD, que, por sua vez, trata da comunicação com o banco de dados. Damas cita, como requisitos fundamentais a um SGBD, eficiência, robustez, controle de acessos e persistência (2007, p. 33).

Existem diversas ferramentas para gerência de bancos de dados disponíveis no mercado. Alguns exemplos são MySQL, PostgreSQL, Oracle DB, Microsoft SQL Server e SQL Lite. Cada aplicação apresenta peculiaridades, mas a essência se mantém a mesma (modelo relacional ou objeto-relacional e uso da linguagem SQL).

Um dos bancos utilizados em sistemas web PHP é o MySQL. Esse SGBD surgiu em 1996, como um projeto interno da empresa TcX DataKonsult AB, na Suécia. Rapidamente, tornou-se muito popular pois, embora não tivesse alguns recursos que outros SGBDs ofereciam, primava pela escalabilidade e bom desempenho, qualidades consideradas mais importantes por muitos. Seu crescimento fez com que fosse adquirido pela Sun Microsystems que foi, pouco depois, comprada pela Oracle Corporation. Atualmente, muitos dos recursos que lhe faltavam foram acrescentados (GILMORE, 2010, 477).

Segundo Gilmore, até a data em que seu livro foi escrito, o MySQL já contava com mais de 100 milhões de *downloads* (2010, p. 477). A lista de clientes é grande, incluindo empresas bem conhecidas como Adobe, globo.com, Walmart, Drupal (sistema de gerenciamento de conteúdo escrito em PHP), entre outras (MySQL CUSTOMERS, 2014).

Gilmore cita duas qualidades que, na sua opinião, ajudaram a popularizar o MySQL. A primeira delas é a flexibilidade, isto é, funciona em inúmeras plataformas e sistemas operacionais diferentes, com APIs (o conceito de API é explicado mais abaixo) disponíveis

para todas as linguagens de programação mais utilizadas. A segunda característica é o seu "poder", em especial no que diz respeito à elevada performance (2010, p. 477 e 478).

O SGBD MySQL foi escolhido para o desenvolvimento dessa pesquisa. No entanto, a partir da aquisição da Sun Microsystems pela Oracle, uma cisão ocorreu. O MySQL permaneceu como propriedade da empresa. Preocupados com os rumos que a nova proprietária tomaria em relação ao banco, no entanto, seus desenvolvedores originais (incluindo o próprio fundador do MySQL, Michael Widenius) abandonaram a empresa e criaram, a partir de seu código fonte, o banco MariaDB (MARIADB, 2014).

O MariaDB mantém compatibilidade quase que total com o MySQL sendo, portanto, possível utilizá-lo com as APIs para MySQL da linguagem PHP (MARIADB IS, 2014).

Em função de que os desenvolvedores originais do MySQL agora trabalham no projeto MariaDB, e como forma de apoio ao software livre, a presente pesquisa fez uso da implementação MariaDB para a elaboração do sistema de *login* e execução dos testes.

3.3 APIS PARA CONEXÃO COM BANCO DE DADOS E PDO

Segundo a documentação da linguagem PHP, uma API (o termo, em inglês, é um acrônimo para *Application Programming Interface*, ou Interface para Programação de Aplicativos) tem a função de definir "classes, métodos, funções e variáveis que sua aplicação necessitará chamar para realizar a tarefa desejada". Em PHP, as APIs para comunicação com bancos de dados são providenciadas, em geral, por meio de extensões, ou seja, código cuja função é estender as funcionalidades programadas no núcleo da linguagem (OVERVIEW, 2014, tradução nossa).

Para conexão com banco de dados MySQL ou MariaDB, a linguagem oferece três APIs: as extensões MySQL, mysqli e PDO. A extensão MySQL já está defasada, e seu uso não é mais recomendado para tecnologias atuais (OVERVIEW, 2014).

A extensão mysqli é uma atualização da MySQL, trazendo a opção de uso de uma interface orientada a objetos, entre vários outros avanços (OVERVIEW, 2014).

A PDO, cujo nome é um acrônimo para PHP *Data Objects* (em português, objetos de dados da PHP), é uma extensão um pouco diferente. Ela provê uma camada de abstração ao acesso aos dados, ou seja, as mesmas funções são usadas de maneira independente do sistema de gerência de banco de dados utilizado. Isso quer dizer que, além de prover acesso a bancos MySQL, a PDO pode trabalhar com outros também, necessitando um software (*driver*) específico para cada tipo de banco (INTRODUCTION, 2014).

A documentação ressalta, no entanto, que a PDO não provê abstração de banco de dados, isto é, não reescreve SQL. Se houver diferenças de sintaxe entre a SQL de um SGBD e outro, essas mudanças terão que ser realizadas manualmente no código (INTRODUCTION, 2014).

A conexão com o banco de dados nas APIs PDO e *mysqli* se dá de maneira similar. Na primeira, instancia-se um objeto PDO, passando ao construtor, como parâmetro, os dados da conexão e, opcionalmente, usuário e senha. Um exemplo do procedimento: `$pdo = new PDO('mysql:host=localhost;dbname=test', 'user', 'password');` (CONNECTIONS AND, 2014). No *mysqli*, *host* e banco são informados separadamente: `$mysqli = new mysqli('localhost', 'user', 'password', 'database');` (CONNECTIONS, 2014).

A API *mysqli* oferece, tal qual a PDO, bons recursos de segurança. No entanto, seu uso está limitado ao banco MySQL. Chris Snyder, Thomas Myer e Michael Southwell mencionam a possibilidade de utilização de bibliotecas externas, como a *PearDB*, para suprir tal limitação. Os autores destacam, no entanto, alguns aspectos negativos do uso de bibliotecas externas: grande aumento na quantidade de código a ser gerenciado, além do fato de que seu uso põe o desenvolvedor à mercê de ideias de outrem a respeito de como implementar aspectos do sistema (2010, p. 42). Com o advento da PDO, tem-se, na própria especificação da linguagem PHP, um recurso de segurança nativo compatível com diversos SGBDs.

Tanto a PDO como a *mysqli* fazem uso de *prepared statements* para realizar consultas ao banco (tal recurso é providenciado pelos *drivers* de SGBDs ou pela própria extensão). *Prepared statements* são comandos SQL pré-compilados para os quais pode-se passar parâmetros com diferentes valores (PREPARED, 2014). Com a PDO, pode-se, por exemplo, criar um *prepared statement* conforme a seguir: `$stmt = $pdo->prepare('INSERT INTO pessoa (nome) VALUES (:nome)')`. Em seguida, atrela-se o parâmetro da *query* a uma variável: `$stmt->bindParam(':nome', $nome)`. Por fim, pode-se atribuir valor à variável e executar o comando: `$nome = 'Victor'; $stmt->execute();`. Com o *mysqli*, o processo é semelhante, com a desvantagem de que não se pode atribuir nomes aos parâmetros. Em lugar de `:nome`, portanto, seria usado o caractere `?`.

A documentação da PHP cita duas vantagens importantes no uso de *prepared statements*: desempenho (um comando só precisa ser preparado pelo SGBD uma vez – passando pelo processo natural de análise, compilação e otimização da consulta –, podendo, em seguida, ser utilizado inúmeras vezes) e segurança (PREPARED, 2014).

Quanto à segurança, o uso de *prepared statements* faz com que os valores entrados no

sistema pelo usuário sejam enviados ao banco, na forma de parâmetros, separadamente em relação às consultas (já pré-compiladas). Assim, tais valores somente são usados em tempo de execução (e não no momento da montagem da consulta), resguardando o sistema contra injeções de SQL. Conforme documentação da Oracle, o uso de *prepared statements* fornece o mesmo nível de segurança em relação a qualquer sistema que formate de maneira correta os dados a serem usados nas consultas, indo além de simplesmente escapar caracteres (ESCAPING, 2014).

4 SEGURANÇA EM SISTEMAS WEB

Para Snyder, Myer e Southwell, a segurança de computadores envolve três aspectos principais. O primeiro diz respeito a manter certas informações secretas (proteger o acesso a dados sigilosos como, por exemplo, informações de cartões de crédito). O segundo aspecto refere-se a recursos escassos: é preciso garantir o acesso dos usuários aos recursos do sistema, impedindo o esgotamento deles, seja ele intencional ou não. Por fim, há o fator Internet: para máquinas conectadas na grande rede, questões antes locais passam a adquirir dimensões muito maiores (2010, p. 3).

No que diz respeito à Internet, existe um agravante que deve ser considerado: usuários mal-intencionados conseguem, ao menos inicialmente, agir de maneira anônima (não se pode checar a real fonte de entradas de dados). A isso soma-se o fato de que é impossível prover completa segurança a um sistema:

> Em um pequeno sistema, pode ser possível descobrir e combater todas as formas possíveis de ataque, ou verificar cada bit. Mas em um sistema operacional moderno, que consiste de muitos processos executando, simultaneamente, centenas de megabytes ou até gigabytes de código e dados, a segurança absoluta está fadada a ser um objetivo, não uma meta alcançável. (SNYDER, MYER e SOUTHWELL, 2010, p. 4, tradução nossa)

No entanto, conforme ressaltam Snyder, Myer e Southwell, a linguagem PHP apresenta bons recursos relacionados à segurança (2010, p. 4, tradução nossa).

Outro conceito importante, nesse contexto, é o de vulnerabilidade. Em uma aplicação, uma vulnerabilidade se refere a um “buraco” – uma fraqueza –, advindo de problemas de *design* ou implementação, que pode ser explorado, por um usuário mal-intencionado, para causar dano ao sistema (e, consequentemente, a todas as pessoas, empresas, organizações, etc., que dele fazem uso) (CATEGORY, 2014).

Isso leva à questão: de onde vêm as vulnerabilidades de sistemas web e quais as formas de ataque a que eles estão vulneráveis? Snyder, Myer e Southwell abordam três tipos principais de vulnerabilidades. A primeira surge quando o usuário tem a possibilidade de entrar com informações no sistema. Nesse sentido, podem ocorrer ataques humanos (executados por pessoas) ou automatizados (quando o poder de computadores é usado para amplificar ataques) (2010, p. 4-7).

O segundo tipo ocorre quando, ao contrário, informações são passadas da aplicação para o usuário. Usuários mal-intencionados podem tentar obter informações relevantes nos

web sites de empresas, visando planejar um ataque. Ao invadir sistemas, podem, também, obter informações a respeito dos usuários cadastrados (SNYDER, MYER e SOUTHWELL, 2010, p. 8).

A última categoria proposta pelos autores é a genérica “outros casos”. Refere-se a tentativas de ataques contra web sites no nível de rede (SNYDER, MYER e SOUTHWELL, 2010, p. 8).

A presente pesquisa está mais centrada nos ataques realizados no âmbito do sistema (programação) e banco de dados, abordando, portanto, as duas categorias anteriores. Em particular, serão abordados os ataques por injeção de código (em especial, referindo-se à linguagem SQL), cuja definição é explorada a seguir.

4.1 ATAQUES POR INJEÇÃO DE CÓDIGO

Injeção de código é um termo genérico que se refere a ataques nos quais um usuário insere, por algum meio, o código que será interpretado/executado pela aplicação (CODE, 2013).

Cabe, aqui, uma distinção: injeção de código é diferente de injeção de comando, pois na primeira “um atacante está limitado somente pela funcionalidade da linguagem injetada em si. Se um atacante é capaz de injetar código PHP em uma aplicação e fazê-lo ser executado, ele é limitado somente pelo que o PHP é capaz de fazer” (CODE, 2013, tradução nossa).

A injeção de comando, por sua vez, caracteriza-se pela utilização de vulnerabilidades na aplicação para executar comandos no sistema atacado (sistema operacional, SGBD, entre outros).

4.2 ATAQUES POR INJEÇÃO DE SQL

Caracterizam-se como um caso particular – uma categoria – dos ataques por injeção de código. Conforme abordado anteriormente, a SQL é a linguagem que permite realizar operações de consulta e manipulação de dados em todos os bancos de dados relacionais. Relaciona-se, portanto, com a parte mais importante de um sistema: as informações nele armazenadas. A maior exposição de sistemas disponíveis através da Internet torna ainda mais essencial o estudo dessa forma de ataque e os procedimentos de prevenção relacionados a ela.

Segundo o *Open Web Application Security Project* (OWASP)[1], uma injeção de SQL pode ser definida como um ataque que consiste na inserção de código SQL na aplicação, de modo que ele seja executado pelo SGBD (SQL, 2014, tradução nossa).

Em muitos casos, em aplicações web, sentenças SQL são construídas unindo comandos codificados pelo desenvolvedor com informações entradas pelo usuário. Diante disso, torna-se possível, para o usuário, inserir código SQL, fazendo com que o SGBD execute comandos não previstos quando do desenvolvimento do sistema (TESTING, 2014).

Em concordância com o OWASP, os autores Halfond, Viegas e Orso definem injeção de SQL como "uma classe de ataques por injeção de código na qual informação fornecida pelo usuário é incluída em uma consulta SQL, de tal forma que parte dos dados entrados pelo usuário são tratados como código SQL" (2006, tradução nossa).

A proporção do problema deve ser bem compreendida: segundo Halfond, Viegas e Orso, vulnerabilidades a injeções de SQL podem fornecer a um usuário mal-intencionado acesso completo ao banco de dados subjacente ao sistema, podendo ocasionar perda ou corrompimento de informações confidenciais, ou até mesmo corrompimento do sistema que hospeda a aplicação (2006).

Ainda, segundo o OWASP, um ataque dessa forma pode permitir a obtenção de dados armazenados no banco, bem como a modificação dessas informações, "a execução de operações administrativas no banco de dados" e até mesmo a execução de operações no sistema operacional (SQL, 2014, tradução nossa).

4.2.1 Formas de ataques por injeção de SQL

O OWASP oferece uma classificação de injeções de SQL que as dividem em três classes principais. Nas chamadas *inband* (em banda), a informação obtida do banco, por meio de uma injeção, é apresentada na própria página web. A *out-of-band* (fora da banda) é a categoria em que o ataque obtém dados através de outro meio (por exemplo, "um e-mail com o resultado da consulta é gerando e enviado" ao usuário). Por fim, existem as injeções inferenciais ou cegas (do inglês, *blind injections*), nas quais "não há real transferência de dados", e o autor das injeções procura inferir informações a respeito do sistema enviando requisições e observando seu comportamento (TESTING, 2014, tradução nossa).

A entidade também descreve cinco técnicas comuns, utilizadas para a confecção de

---

1 A OWASP é uma fundação sem fins lucrativos que existe desde dezembro de 2001, e cujo objetivo é auxiliar organizações na criação de aplicações seguras e confiáveis. Tornou-se uma importante referência na documentação de formas de ataque, testes e prevenção no que diz respeito à segurança de sistemas web.

injeções de SQL. A primeira refere-se ao uso do comando UNION, com o qual é possível explorar brechas na geração de comandos SELECT no sistema. Tal comando permite "combinar duas consultas em um único conjunto de resultados", em determinadas situações (TESTING, 2014, tradução nossa).

A segunda técnica consiste no uso de condições *Booleanas*, visando a descobrir se determinada condição é ou não verdadeira. Em seguida, há a técnica baseada em erro, cujo objetivo é provocar, deliberadamente, consultas que não sejam executadas com êxito, de modo a descobrir informações sobre o sistema com base nos erros gerados. A última técnica, conhecida como *time delay* (atraso), faz uso de comandos como o SLEEP, em consultas condicionais, de modo a descobrir, em um sistema que não fornece muitas informações, se determinada condição retornou verdadeiro ou falso (TESTING, 2014).

Todas as técnicas citadas podem ser usadas em uma grande variedade de injeções. Halfond, Viegas e Orso também fornecem uma classificação, catalogando os tipos de injeções de SQL utilizados e exemplificando seu uso (2006).

A primeira categoria destacada pelos autores, e uma das formas mais simples de injeções, é a tautologia. Como o próprio nome indica, ela objetiva injetar código "em uma ou mais sentenças condicionais de modo que elas sejam sempre avaliadas para *true*. (HALFOND, VIEGAS e ORSO, 2006, tradução nossa).

Um uso comum da tautologia é tentar enganar a autenticação de um sistema, por exemplo, em um formulário de *login*. Halfond, Viegas e Orso (2006) fornecem um exemplo: o usuário poderia digitar os caracteres `' or 1 = 1 --`. A consulta enviada ao banco está representada na figura 11.

Figura 11: Injeção de tautologia: a comparação "1=1" resultará sempre verdadeira

```
SELECT accounts
  FROM users
  WHERE login='' or 1=1
    -- AND pass='' AND pin=''
```

Fonte: HALFOND, VIEGAS e ORSO, 2006

No exemplo acima, a injeção foi digitada no campo de *login*. Note-se que a aspa simples (') fecha a *string* de login; "or 1=1" é, obviamente, a tautologia. Os caracteres "--" comentam o restante da consulta. Como 1 é igual a 1, o resultado da avaliação será sempre *true*, permitindo ao usuário logar-se no sistema sem conhecer credenciais reais.

A segunda categoria refere-se às consultas "logicamente incorretas". Seu objetivo é,

simplesmente, gerar consultas incorretas, visando obter informações a partir de erros gerados pelo banco ou sistema. Costuma ser usada como um ataque preliminar, para obter informações e criar possibilidades de ataques futuros (HALFOND, VIEGAS e ORSO, 2006).

Essa injeção se baseia no fato de que, frequentemente, o servidor, diante de um erro gerado pela injeção, revela importantes informações a respeito das tecnologias utilizadas (como o SGBD, por exemplo). No artigo, também consta um exemplo dessa forma de ataque, ilustrado na figura 12.

Figura 12: Injeção que visa gerar erros no banco

```
SELECT accounts
  FROM users
  WHERE login=''
    AND pass=''
    AND pin= convert (int, (select top 1 name
                            from sysobjects
                            where xtype='u'))
```

Fonte: HALFOND, VIEGAS e ORSO, 2006

Note-se que a tentativa de converter um valor textual para um inteiro ocasionará o erro (HALFOND, VIEGAS e ORSO, 2006).

Um terceiro tipo de injeção é a chamada consulta de união. Consiste na inserção de uma sentença de união (utilizando o operador UNION) (HALFOND, VIEGAS e ORSO, 2006). O operador UNION permite unir uma consulta em outra para obter outros dados além dos resultados da consulta original que o sistema faz (TESTING 2014).

A OWASP provê um exemplo que esclarece o funcionamento dessa forma de injeção. Se um sistema realiza a consulta `SELECT Name, Phone, Address FROM Users WHERE Id=$id`, e conseguirmos injetar uma sentença UNION, gerando a consulta `SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable`, será possível obter as informações de todos os números de cartões de créditos, armazenados na tabela *CreditCardTable*. Além do número do cartão, mais dois valores – o número 1 duas vezes – são adicionados à tabela de resultado ("creditCardNumber,1,1"), pois o comando UNION exige que as consultas unidas retornem o mesmo número de colunas (TESTING, 2014).

O quarto tipo, segundo a categorização fornecida por Halfond, Viegas e Orso, é a consulta combinada (tradução nossa para o termo "piggy-backed query"). Seu objetivo é extrair informações ou modificá-las, ou ainda promover negação de serviço ou executar comandos remotos (2006).

Essa categoria apresenta, como ressaltam os autores, uma importante diferença em relação a todas as outras formas de injeção de SQL: em lugar de tentar modificar a consulta original da aplicação, ela procura injetar uma consulta adicional. Se o usuário mal-intencionado conseguir encontrar uma vulnerabilidade para uma consulta combinada, as possibilidades de injeções, e danos ao sistema, serão muito grandes (HALFOND, VIEGAS e ORSO, 2006).

No exemplo fornecido pelos autores, o usuário injeta a *string* `';drop table users--` no campo senha de um formulário de *login*. Se a entrada de dados não for validada, a consulta resultante será `SELECT accounts FROM users WHERE login='doe' AND pass=''; drop table users -- ' AND pin=123"`. Ela ocasionará, então, a deleção da tabela *users* do banco de dados (2006).

Note-se que somente os bancos de dados configurados para aceitar a execução de mais de uma consulta em uma mesma *string* de entrada estão vulneráveis a essa forma de ataque (HALFOND, VIEGAS e ORSO, 2006).

Outra classe de injeção é a de *stored procedures*. Consiste na tentativa de injetar SQL em rotinas de código executadas pelo SGBD (chamadas *stored procedures*). Segundo Halfond, Viegas e Orso, muitos acreditam que o uso de *stored procedures* torna um sistema web imune a injeções de SQL. Essa crença é falsa, uma vez que os procedimentos armazenados no banco podem ser tão vulneráveis quanto rotinas programadas na linguagem do sistema. A injeção de SQL, se a entrada não for validada, poderá ser executada normalmente, ainda que dentro de um *stored procedure* (2006).

Os autores propõem, como exemplo, um *stored procedure* responsável pela autenticação de um usuário, conforme a figura 13.

Figura 13: *Stored procedure* que autentica um usuário

```
CREATE PROCEDURE DBO.isAuthenticated
    @userName varchar2, @pass varchar2, @pin int
AS
    EXEC("SELECT accounts FROM users
    WHERE login='" +@userName+ "' and pass='" +@password+
          "' and pin=" +@pin);
GO
```

Fonte: HALFOND, VIEGAS e ORSO, 2006

No exemplo, o usuário injeta o código `' ; SHUTDOWN; - -` no campo de *login* ou senha de um formulário de autenticação. Como resultado, dentro do *stored procedure*, o comando SHUTDOWN é disparado, fazendo com que o SGBD pare de rodar (HALFOND,

VIEGAS e ORSO, 2006).

Outra forma de injeção de SQL é a inferência, cujo objetivo é "identificar parâmetros injetáveis, extrair informações" ou "determinar o esquema de banco de dados". Para levar a cabo esse tipo de ataque, é preciso modificar a consulta, de modo a transformá-la em uma ação que será executada com base na resposta a uma questão *true/false* a respeito de informações contidas no banco de dados. É frequentemente usada quando o usuário mal-intencionado pretende adquirir informações que possam auxiliar em ataques futuros, mas o sistema e SGBD foram configurados para não mostrar tais informações ao usuário (HALFOND, VIEGAS e ORSO, 2006, tradução nossa).

As injeções do tipo inferência podem ser subdivididas em duas categorias: injeções cegas (do inglês, *blind injections*), e as baseadas em tempo (do inglês, *timing attacks*). Nas injeções cegas, tenta-se inferir informações sobre o sistema a partir do comportamento da página, perguntando ao SGBD questões *true/false*. Mesmo que não haja mensagens de erro, quando a consulta retorna *false*, há mudanças no comportamento da página (HALFOND, VIEGAS e ORSO, 2006).

Já nos ataques baseados em tempo, o usuário mal-intencionado observa atrasos na resposta do SGBD: faz uso de instruções do tipo *if/then* e construtos SQL que levam um certo tempo para sua execução (por exemplo, o palavra-chave WAITFOR). Medindo o tempo de resposta, o atacante consegue saber que parte de sua injeção foi executada e, assim, a resposta para a questão *true/false* (HALFOND, VIEGAS e ORSO, 2006).

Halfond, Viegas e Orso provêm um exemplo de cada um desses tipos de injeção. Na injeção cega, o usuário poderia inserir, no campo de login, duas injeções, uma de cada vez. Na primeira, digitaria `legalUser' and 1=0 --`, gerando a consulta mostrada na figura 14.

Figura 14: Primeira etapa de uma injeção cega

```
SELECT accounts
  FROM users
  WHERE login='legalUser'
    AND 1=0 -- ' AND pass='' AND pin=0
```

Fonte: HALFOND, VIEGAS e ORSO, 2006

Em seguida, digita-se, no campo de login, `legalUser' and 1=1 --`, gerando a consulta expressa na figura 15.

Figura 15: Segunda etapa de uma injeção cega

```
SELECT accounts
  FROM users
  WHERE login='legalUser'
    AND 1=1 -- ' AND pass='' AND pin=0
```

Fonte: HALFOND, VIEGAS e ORSO, 2006

Se, em ambos os casos, houver uma mensagem de erro de login, então o campo está protegido contra injeção. Caso contrário, a segunda tentativa não retornará erro, já que é uma tautologia (1=1 sempre retorna *true*), e pode-se inferir que o campo de login está vulnerável (HALFOND, VIEGAS e ORSO, 2006).

No exemplo fornecido para ataques baseados em tempo, o usuário digita, no parâmetro de *login*, a *string* `''legalUser' and ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) > X WAITFOR 5 --''`, gerando a consulta da figura 16.

Figura 16: Injeção baseada em tempo, fazendo uso da instrução WAITFOR

```
SELECT accounts
  FROM users
  WHERE login='legalUser'
    AND ASCII(SUBSTRING(
      (select top 1 name from sysobjects),1,1)) > X WAITFOR 5
      -- ' AND pass='' AND pin=0
```

Fonte: HALFOND, VIEGAS e ORSO, 2016

Nesse caso, se o valor ASCII[2] do primeiro caractere do campo *name*, na tabela *sysobjects* (tabela de *meta-dados* do banco Microsoft SQL Server), for maior que o valor de X, haverá um atraso de 5 segundos, causado pelo comando WAITFOR. Com isso, é possível, variando o valor de X, determinar qual é o caractere na tabela (HALFOND, VIEGAS e ORSO, 2006).

A última categoria de ataque é a baseada em codificações alternativas. Nela, o texto injetado é modificado, de modo a evitar detecção por códigos defensivos do sistema, bem como ludibriar técnicas que visam prevenir injeções. É naturalmente usado conjuntamente com outros tipos de injeção (HALFOND, VIEGAS e ORSO, 2006).

Halfond, Viegas e Orso fornecem um exemplo em que o usuário digita `legalUser'; exec(0x73687574646f776e) --` no campo de login, resultando na consulta mostrada na figura 17.

2 ASCII: do inglês, *American Standard Code for Information Interchange*, ou Código Padrão Americano para o Intercâmbio de Informação. Trata-se de um tipo de codificação para representação de caracteres em computadores.

Figura 17: Injeção baseada em codificações alternativas

```
SELECT accounts
  FROM users
  WHERE login='legalUser'; exec(char(0x73687574646f776e))
    -- AND pass='' AND pin=
```

Fonte: HALFOND, VIEGAS e ORSO, 2006

Essa injeção faz uso de codificação ASCII hexadecimal. A função *char()* recebe, como parâmetro, a codificação inteira ou hexadecimal de um caractere e retorna uma instância dele. Como a sequência de caracteres hexadecimais acima representa, na codificação ASCII hexadecimal, a *string* "SHUTDOWN", o comando de desligar será executado no banco, se a injeção funcionar (HALFOND, VIEGAS e ORSO, 2006).

4.2.2 Prevenção de ataques por injeção de SQL

A origem de ataques por injeção de SQL é a entrada de dados efetuada por um usuário do sistema. Portanto, uma forma comum de tentar prevenir injeções é a validação dessas informações. Duas abordagens podem ser utilizadas: aceitar apenas caracteres que sejam parte de uma "lista branca" de valores aceitos; ou procurar o texto entrado pelo usuário e "escapar" caracteres potencialmente maliciosos aí presentes. Eliminar caracteres que poderão causar problemas não garante a total segurança de um sistema, embora possa evitar várias formas de injeção (SQL, 2014).

Halfond, Viegas e Orso referem-se ao uso da "lista branca" como uma forma de validação "positiva", uma vez que procura identificar entradas de dados válidas, em lugar de procurar por caracteres potencialmente maliciosos ("validação negativa"). Segundo os autores, a primeira abordagem é superior, uma vez que é muito difícil tentar prever todas as formas de injeções que podem ser inseridas em um dado campo, sendo mais fácil conhecer e checar o tipo de dado que deve ser aceito naquela situação. Ressaltam, ainda, a importância de identificar todas as fontes de entrada de dados do sistema. Qualquer campo que receba informações do usuário e use esses dados na construção de consultas pode gerar brechas para injeções de SQL (HALFOND, VIEGAS e ORSO, 2006).

Outra proposição, verdadeira apenas em parte, é a de que o uso de *stored procedures*, por si só, representa uma defesa contra injeções de SQL. Conforme ressalta a OWASP, *stored procedures*, conquanto sejam úteis na prevenção de algumas formas de ataque, são vulneráveis a muitas outras (SQL, 2014). O estudo realizado por Halfond, Viegas e Orso

chegou às mesmas conclusões expressas no documento da OWASP: *stored procedures* podem apresentar vulnerabilidades assim como o código programado na aplicação. (2006).

Regras simples e bastante conhecidas de boas práticas em programação de sistemas web podem dificultar bastante a vida de usuários mal-intencionados. Uma página que, diante de um erro ocorrido em uma consulta ao banco (por vezes provocado intencionalmente, por meio de uma injeção de SQL), mostra dados a respeito do SGBD e da consulta ao usuário, pode acabar auxiliando na construção de injeções de SQL que venham a ser executadas com sucesso. Páginas que escondem erros técnicos, por outro lado, requerem que o atacante faça uso de engenharia reversa para descobrir a lógica da consulta original, dificultando sua ação (TESTING, 2014).

4.3 TESTES

Testar um sistema em relação a vulnerabilidades para injeções de SQL implica conhecer os momentos em que isso pode ocorrer: ocasiões nas quais o sistema necessita interagir com o usuário e o banco (TESTING, 2014).

A OWASP fornece três exemplos em que tal interação ocorre: "formas de autenticação" (por exemplo, um sistema de *login*, em que nome de usuário e senha são digitados em um formulário, para que tais informações sejam comparadas com as presentes no banco); motores de busca; e sites *E-Commerce* (TESTING, 2014).

Um dos primeiros passos para o teste de um sistema é procurar as formas pelas quais as informações podem ser inseridas no sistema por usuários para serem utilizadas em consultas. A recomendação da OWASP é fazer uma lista de tais possibilidades e, em seguida, testá-las, inserindo informações que possam interferir nas consultas originais. É importante frisar que as vulnerabilidades não existem apenas em campos de formulários e na URL (em requisições GET): é preciso observar "campos escondidos de requisições POST", bem como "cabeçalhos HTTP e *Cookies*" (TESTING, 2014, tradução nossa).

Alguns testes bastante rápidos e simples já podem identificar algumas das vulnerabilidades mais básicas. Pode-se inserir, no campo testado, uma aspa simples (') ou um ponto e vírgula (;). A aspa simples atua como um delimitador de *strings* em SQL, e o ponto e vírgula determina o final de um determinado comando. Dessa forma, se o sistema não filtrar a entrada de tais caracteres, um erro pode ser gerado (TESTING, 2014, tradução nossa).

Para tornar uma injeção mais efetiva, é importante conhecer o SGBD a ser atacado. Nesse contexto, um usuário mal-intencionado tentará descobrir o tipo de sistema de

gerenciamento de banco de dados e sua versão a partir dos erros que a aplicação retorna. Portanto, executar testes com o objetivo de gerar erros na aplicação, observando se informações valiosas estão sendo fornecidas nas páginas web, também é uma etapa importante (TESTING, 2014).

Encontrar vulnerabilidades e prevenir ataques é uma questão complexa e, por isso, não há apenas uma resposta. No entanto, é vital procurar encontrar respostas, ainda que parciais, para o problema. Cada vez mais, informações importantes precisam ser compartilhadas e, dessa forma, protegê-las contra acessos e alterações indevidos é uma etapa crucial no planejamento e implementação de sistemas em rede.

4.3.1 Ferramentas para testes e SQLMap

Muitos dos testes que devem ser executados para verificar o nível de segurança de um sistema podem ser realizados de maneira automatizada, através do uso de ferramentas específicas para isso.

A SQLMap é um aplicativo de linha de comando, gratuito e livre, cujo propósito é auxiliar na detecção de vulnerabilidades a ataques por injeção de SQL. Ela permite detectar, automaticamente, “parâmetros vulneráveis”, “identificar quais técnicas de injeções de SQL podem ser usadas para explorar” tais parâmetros, extrair informações a respeito do SGBD e até mesmo assumir total controle sobre ele (INTRODUCTION, 2014). Pode ser utilizada, dessa forma, para obter maior compreensão a respeito das vulnerabilidades a injeções de SQL presentes em um sistema, bem como os riscos que elas representam se exploradas.

Conforme a documentação oficial, a ferramenta oferece suporte aos seguintes tipos de SGBD: MySQL, Oracle, PostgreSQL, Microsoft SQL Server, Microsoft Access, IBM DB2, Firebird, Sybase, SAP MaxDB e HSQLDB (FEATURES, 2014). Além disso, ela é capaz de executar testes com base em cinco tipos de injeções: injeções por inferência (cegas e baseadas em tempo), injeções baseadas em consultas logicamente incorretas, consultas de união, injeção de consulta de união parcial (quando a aplicação retorna apenas o primeiro resultado encontrado), e injeções por consulta combinada (*piggy-backed*) (TECHNIQUES, 2014).

Devido a seu bom suporte a diferentes tipos de injeções e à maior precisão que uma ferramenta especializada é capaz de oferecer, no presente trabalho, optou-se pelo uso da SQLMap para execução da maior parte dos testes para detecção de vulnerabilidades.

A ferramenta funciona por meio de uma interface por linha de comando, e diversas opções podem ser utilizadas para determinar seu comportamento. A figura 18 exemplifica sua

utilização:

Figura 18: Exemplo de uso da ferramenta SQLMap, passando uma URL como parâmetro

```
python sqlmap.py -u "http://www.target.com/vuln.php?id=1" -f --banner --dbs --users
```

Fonte: USAGE, 2014

No exemplo acima, utilizou-se a opção -u para especificar a URL a ser testada. A opção -f indica que se quer realizar uma análise detalhada do SGBD, obtendo o máximo de informações possível. Algumas opções de enumeração também foram utilizadas: -dbs (para listar os bancos de dados existentes no SGBD), --users (para enumerar os usuários presentes) e, por fim, --banner (para obter tipo e versão do SGBD). Note-se que um parâmetro é passado na própria URL (*id*), de modo a verificar a presença de vulnerabilidades a injeções de SQL.

Os comandos e opções utilizados para os objetivos da presente pesquisa são melhor detalhados no capítulo 5, que trata especificamente da implementação de um sistema de *login* e testes de vulnerabilidades a ataques por injeção de SQL.

5 IMPLEMENTAÇÃO E EXECUÇÃO DE TESTES

Este capítulo é dedicado a descrever os testes de ataques por injeção de SQL realizados sobre um sistema de *login* feito com a linguagem de programação PHP e a extensão para conexão com banco de dados PDO. O sistema foi desenvolvido especificamente para a execução dos testes aqui descritos. Nesta etapa, objetivava-se por a PDO à prova, observando seu comportamento e os níveis de segurança alcançados por meio dela, utilizando-se *prepared statements* nas consultas ao banco.

Para efetuar os ataques por injeção de SQL ao sistema, foram usadas, basicamente, duas estratégias: alguns ataques foram realizados manualmente, por meio da injeção de campos do formulário de *login*; a seguir, testes mais abrangentes foram conduzidos com a utilização de uma ferramenta específica: a SQLMap.

Os testes foram realizados em duas etapas. Na primeira, as injeções manuais e a SQLMap foram testadas contra o sistema sem a utilização de *prepared statements.* A consulta ao banco utilizada na autenticação do usuário foi montada diretamente, concatenando as informações do formulário de *login* à *string* contendo o comando SELECT. Em seguida, a aplicação foi modificada para utilizar *prepared statements*, e os dados digitados pelo usuário nos campos do formulário foram parametrizados.

Visando melhor averiguar a eficácia do uso de *prepared statements* na prevenção contra ataques por injeção de SQL, nenhuma outra medida foi deliberadamente tomada para aumentar a segurança do sistema na segunda etapa dos testes.

5.1 IMPLEMENTAÇÃO DO SISTEMA DE *LOGIN*

Utilizou-se, para o sistema, o banco de dados MariaDB, contendo apenas uma tabela de usuários, formada pelas colunas id (chave primária), *username* (nome de usuário) e *password* (senha). As duas últimas colunas contêm os dados a serem usados na autenticação do usuário.

O primeiro passo para implementar o sistema de *login* usado para os testes foi criar classes para tratar da conexão com o banco de dados, mapeando a relação entre o banco e os objetos do sistema.

Utilizou-se o padrão de projeto “Data Mapper”, conforme definido por Zandstra (2010). O objetivo desse padrão é estabelecer um mapeamento objeto-relacional. Uma classe principal, denominada Mapper, trata de operações gerais, delegando a classes específicas o

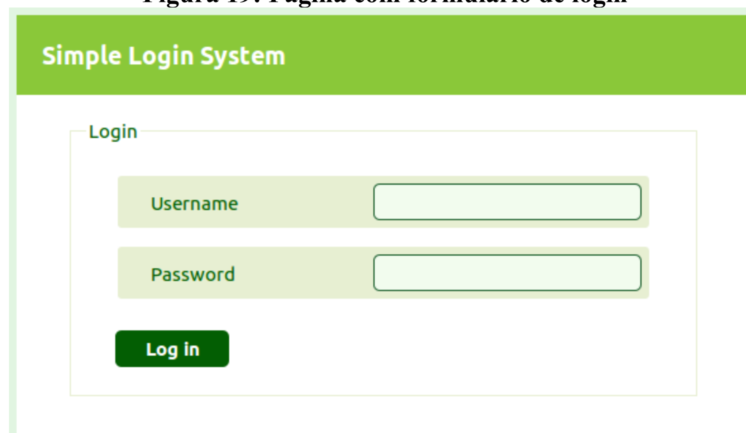
mapeamento de entidades do sistema (por exemplo, uma classe UserMapper mapeia, especificamente, a tabela *users* do banco e a classe User) (ZANDSTRA, 2010, p. 277).

A classe Mapper possui uma instância da classe PDO como atributo. Para além disso, implementa o método *find* (que busca informações no banco e retorna um objeto), e delega para as classes que a estenderem demais operações relacionadas ao banco de dados e ao mapeamento objeto-relacional.

Foi criada, também, uma classe User, para representar o usuário do sistema. Ela contém alguns atributos simples, como nome de usuário e senha, além de um método que verifica se o usuário encontra-se logado no sistema. Contem, ainda, um método responsável por verificar credenciais e efetuar o *login*. Não foi adicionada ao sistema nenhuma validação, e nem mesmo foram filtradas as informações oriundas do formulário (recebidas diretamente através da variável $_POST). Visando simular melhor a autenticação de uma aplicação real, no entanto, fez-se uso de um *hash*[1] na senha, em lugar de salvá-la como texto puro.

O formulário de *login* pode ser observado na figura 19. Ele contém apenas os campos de *Username* e *Password*, além de um botão para submeter as informações.

Figura 19: Página com formulário de login



Fonte: Do autor

Uma vez submetido o formulário, a classe User é instanciada e seu método *login* é chamado. Por meio da classe UserMapper, os dados são buscados no banco para averiguar se as informações fornecidas são válidas. Caso o *login* falhe, uma mensagem de erro é mostrada. Se obtiver sucesso, o usuário é redirecionado para outra página.

A primeira implementação do método *selectUser*, chamado pelo método *login* para obter informações do banco a partir das credenciais digitadas pelo usuário, aparece na figura

1 O mecanismo de *Hashing* transforma senhas de tamanho variável em sequências de caracteres de tamanho fixo, encriptando-as.

20. Note-se que, apesar de a consulta ter sido realizada por meio de um objeto PDO (instanciado na classe UserMapper), não foi utilizada a técnica de *prepared statements*.

Figura 20: Método selectUser implementado sem prepared statements: uma string é montada com o comando SELECT e os dados entrados pelo usuário

```
public function selectUser( $username, $password ) {
    $userMapper = new Mapper\UserMapper();

    $password = Hash::make( $password );
    $sql = "SELECT * FROM users WHERE username = '{$username}' AND password = '{$password}'";

    if ( $rs = $userMapper->rawSelect( $sql ) ) {
        if ( $rs->rowCount() === 1 ) {
            $this->result = $rs->fetch( \PDO::FETCH_OBJ );
            return true;
        }
    }

    return false;
}
```

A bateria de testes foi, então, executada sobre essa versão do sistema de *login*. A descrição dela e dos resultados obtidos encontra-se mais adiante, na próxima seção.

Em seguida, o sistema foi modificado, e o recurso de *prepared statements* foi utilizado para consultar informações no banco (os dados entrados pelo usuário foram devidamente parametrizados). Os testes foram realizados novamente, sobre essa nova implementação do sistema, visando averiguar a eficácia de tal estratégia na prevenção de ataques por injeção de SQL. A figura 21 demonstra um exemplo de *prepared statement* criado a partir de um objeto PDO:

Figura 21: Prepared statement criado a partir de um objeto PDO. A partir disso, o comando preparado pode ser executado com diferentes parâmetros.

```
$this->selectStmt = self::$_pdo->prepare(
    "SELECT * FROM users WHERE username=? AND password=?"
);
```

Fonte: Do autor

Os métodos *selectUser* (em sua nova implementação) e *login* são mostrados na figura 22.

Figura 22: Métodos selectUser e login, na classe User, são responsáveis pela autenticação do usuário no sistema. O método selectUser aparece, aqui, implementado com a utilização de prepared statements e parametrização

```
public function selectUser( $username, $password ) {
    $userMapper = new Mapper\UserMapper();
    $select = $userMapper->selectStmt();
    $select->bindParam( 1, $username, \PDO::PARAM_STR );
    $select->bindParam( 2, Hash::make( $password ), \PDO::PARAM_STR );

    if ( $select->execute() ) {
        if ( $select->rowCount() === 1 ) {
            $this->result = $select->fetch( \PDO::FETCH_OBJ );
            return true;
        }
    }

    return false;
}

public function login( $username = null, $password = null ) {
    if ( $this->selectUser( $username, $password ) ) {
        $_SESSION["user"] = $this->result->id;
        return true;
    } else {
        return false;
    }
}
```

Fonte: Do autor

A seguir, encontram-se descritos os testes realizados sobre as duas implementações do sistema, e os resultados obtidos em cada uma das etapas.

5.2 REALIZAÇÃO DE TESTES DE PENETRAÇÃO

Sobre a importância de testar adequadamente um software para dirimir vulnerabilidades, o guia de testes publicado pela OWASP cita um estudo segundo o qual um terço dos custos ocasionados por programas inseguros (nada menos que 22 bilhões de dólares) poderia ser economizado, anualmente, por meio de testes adequados. O documento define teste como o "processo de comparar o estado de um sistema ou aplicação tomando por base um conjunto de critérios". (MEUCCI e MULLER, 2014, p. 9, tradução nossa).

Para Chris Snyder, Thomas Myer e Michael Southwell, para garantir que um sistema encontra-se protegido contra ataques por injeção de SQL, a melhor maneira é executar testes, tentando efetuar injeções, e observar seu comportamento (2010, p. 42).

Tal prática é denominada teste de penetração. Existem diversas formas de testes de

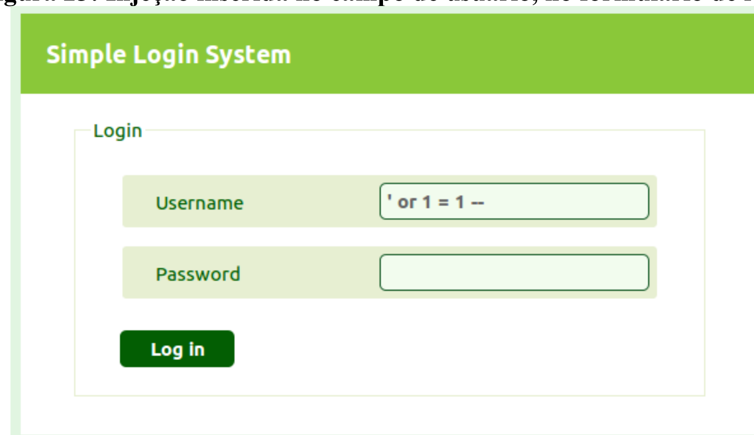
segurança, que podem envolver conversas com desenvolvedores, inspeção do código, entre outras metodologias. A presente pesquisa fez uso, basicamente, de testes de penetração, nos quais o testador simula a atuação de um usuário mal-intencionado, promovendo ataques ao sistema e observando seu comportamento. Testes de penetração apresentam a desvantagem de ocorrerem tarde demais no desenvolvimento do software. Portanto, devem ser utilizados, na atividade profissional, juntamente a outras formas de teste. No entanto, eles servem precisamente os propósitos da presente pesquisa (MEUCCI e MULLER, 2014, p. 14).

5.2.1 Injeções realizadas em sistema sem *prepared statements*

Os primeiros testes, na presente pesquisa, foram realizados no sistema sem utilização de *prepared statements* e parametrização.

Inicialmente, criou-se uma injeção do tipo tautologia, visando realizar *login* no sistema sem a utilização de credenciais reais. No campo de usuário, injetou-se os caracteres `' or 1 = 1 --`, deixando-se o campo de senha vazio. Embora não tenha obtido sucesso em efetuar o *login* (a consulta gerada a partir do objeto PDO não interpretou o comentário que deveria eliminar o restante da *string*), a injeção serviu como uma consulta logicamente incorreta (gerou um erro de SQL), mostrando que o campo de usuário estava vulnerável. As figuras 23 e 24 mostram, respectivamente, a injeção realizada e o resultado obtido.

Figura 23: Injeção inserida no campo de usuário, no formulário de login



Fonte: Do autor

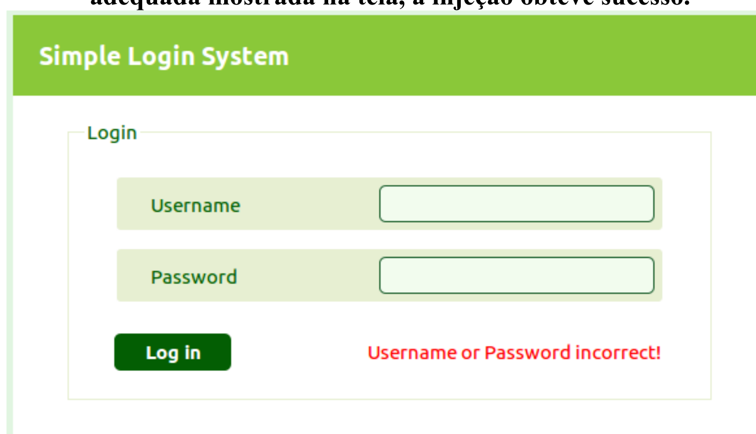
Figura 24: Resultado mostrado na tela: o campo de usuário mostrou-se vulnerável à injeção

Fatal error: Uncaught exception 'PDOException' with message 'SQLSTATE[42000]: Syntax error or access violation: 1064 You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near '' AND password = ''' at line 1' in /home/victor/websites/login-system-tcc-sem-ps/classes/UserMapper.php:36 Stack trace: #0 /home/victor/websites/login-system-tcc-sem-ps/classes/UserMapper.php(36): PDO->query('SELECT * FROM u...') #1 /home/victor/websites/login-system-tcc-sem-ps/classes /User.php(77): classes\mapper\UserMapper->rawSelect('SELECT * FROM u...') #2 /home/victor/websites/login-system-tcc-sem-ps/classes/User.php(88): classes\domain \User->selectUser('' or 1 = 1 --', '') #3 /home/victor/websites/login-system-tcc-sem-ps/login.php(12): classes\domain\User->login('' or 1 = 1 --', '') #4 {main} thrown in **/home/victor/websites/login-system-tcc-sem-ps/classes/UserMapper.php** on line **36**

Fonte: Do autor

Em seguida, foi realizada outra tentativa de injeção no formulário de *login*, desta vez do tipo *piggy-backed*. No campo de usuário, foi digitada a injeção `'; drop table users; --`, e o campo de senha foi deixado novamente vazio. O parâmetro mostrou-se vulnerável a essa forma de ataque. O sistema apenas mostrou uma mensagem de erro de *login*, conforme expresso na figura 25. No entanto, a tabela *users* foi, de fato, removida.

Figura 25: Resposta dada pelo sistema à injeção *piggy-backed*. Apesar da mensagem adequada mostrada na tela, a injeção obteve sucesso.



Fonte: Do autor

A seguir, foram realizados testes com uso da ferramenta SQLMap. Para tanto, o seguinte comando foi utilizado (executado no terminal a partir do diretório contendo os arquivos da SQLMap):

```
python sqlmap.py -v 3 -u \
"http://phpstudies.home/login-system-tcc/login.php" --data \
"submit=1&username=user&password=random"
```

A opção -v (*verbose*) indica o nível de detalhamento das informações mostradas pela ferramenta enquanto executa as ações requisitadas pelo usuário. No nível 3, são mostradas mensagens de *debug*, bem como a carga de dados transmitidos (no inglês, *payload*) (USAGE, 2014).

É preciso, sempre, especificar um alvo para os testes. Nesse caso, utilizou-se a opção -u, que permite especificar uma URL. Pode-se averiguar a presença de vulnerabilidades em parâmetros passados via GET apenas incluindo-os, acompanhados de valores arbitrários, na própria URL. Como, nesse caso, era necessário testar parâmetros POST, utilizou-se a opção --data, seguida dos nomes dos parâmetros e valores arbitrários (USAGE, 2014).

Como pode ser observado na figura 26, o parâmetro *username* mostrou-se vulnerável. O campo de senha não foi injetável em função da conversão do texto digitado para um valor *hash*, prática utilizada para simular um sistema de *login* de uma aplicação real.

Figura 26: A execução da ferramenta SQLMap encontrou vulnerabilidades no parâmetro username

```
[18:29:06] [DEBUG] skipping test 'Generic UNION query (NULL) - 82 to 100 columns' because the leve
l (5) is higher than the provided (1)
[18:29:06] [DEBUG] skipping test 'Generic UNION query (random number) - 82 to 100 columns' because
 the level (5) is higher than the provided (1)
[18:29:06] [DEBUG] checking for filtered characters
POST parameter 'username' is vulnerable. Do you want to keep testing the others (if any)? [y/N]
```

Fonte: Do autor

A SQLMap também conseguiu detectar o tipo de SGBD utilizado na aplicação, bem como o sistema operacional e o servidor subjacentes, informações que poderiam ser utilizadas, por um usuário mal-intencionado, para criar formas de ataque mais específicas.

5.2.2 Injeções realizadas em sistema parametrizado com *prepared statements*

Os testes realizados sobre o sistema modificado para utilizar *prepared statements* e parametrização foram da mesma natureza dos descritos anteriormente.

Inicialmente, injetou-se uma tautologia no campo de nome de usuário: `' or 1 = 1 --`. O campo de senha foi deixado vazio. O sistema mostrou uma mensagem de nome de usuário ou senha incorretos.

Em seguida, tentou-se injetar uma consulta do tipo *piggy-backed*. No campo de usuário, digitou-se a seguinte consulta: `'; drop table users; --`. Novamente o sistema apenas informou que as credenciais estavam incorretas. Ao contrário do teste realizado no sistema não parametrizado, a injeção *piggy-backed* não obteve êxito dessa vez (a tabela *users* não foi removida).

A seguir, foram executados dois testes com a utilização da ferramente SQLMap.

No primeiro, foi utilizado um comando idêntico ao descrito anteriormente:

```
python sqlmap.py -v 3 -u \
"http://phpstudies.home/login-system-tcc/login.php" --data \
```

```
"submit=1&username=user&password=random"
```

Os resultados retornados ao final da execução da ferramenta podem ser vistos na figura 27. Note-se que não foram encontradas vulnerabilidades em nenhum dos campos testados.

Figura 27: Resultado da primeira execução da SQLMap

```
[20:26:13] [WARNING] POST parameter 'password' is not injectable
[20:26:13] [CRITICAL] all tested parameters appear to be not injectable. Try to increase '--level'/'--risk' values to perform more tests. Also, you can try to rerun by providing either a valid value for option '--string' (or '--regexp')

[*] shutting down at 20:26:13
```

Fonte: Do autor

Diante disso, realizou-se um novo teste, fazendo uso da opção --level, à qual atribuiu-se o valor 3, de modo a fazer com que a ferramenta testasse, além dos campos do formulário de login, o cabeçalho HTTP *User-Agent*, bem como o cabeçalho *Referer*. Além disso, foi utilizada a opção --risk, passando-se a ela o valor 3. Ao fazer com que a ferramenta tente realizar injeções com a utilização do operador OR, o risco nível 3 pode ocasionar a atualização de todas as entradas do banco (em comandos UPDATE), caso determinadas tentativas de injeção obtenham sucesso. Ao especificar-se valores maiores para as opções --level e --risk, mais tipos de injeções são testados, ampliando o alcance da bateria de testes (USAGE, 2014). O comando completo é mostrado abaixo:

```
python sqlmap.py -v 3 -u \
"http://phpstudies.home/login-system-tcc/login.php" --data \
"submit=1&username=user&password=random" --level 3 --risk 3
```

Conforme expresso na figura 28, o resultado obtido foi o mesmo do teste anterior.

Figura 28: Informações mostradas no segundo teste executado com a SQLMap

```
[20:33:12] [WARNING] User-Agent parameter 'User-Agent' is not injectable
[20:33:12] [CRITICAL] all tested parameters appear to be not injectable. Try to increase '--level'/'--risk' values to perform more tests. Also, you can try to rerun by providing either a valid value for option '--string' (or '--regexp')

[*] shutting down at 20:33:12
```

Fonte: Do autor

Para a opção --risk, 3 é o maior valor possível. Para --level, pode-se incrementar o

valor até 5. Nesse último nível, uma maior variedade de dados (*payload*) é utilizada nos testes (USAGE, 2014). Mesmo aumentando o valor da opção --level para 5, no entanto, os resultados mantiveram-se inalterados. Nenhuma vulnerabilidade foi detectada no sistema quando as consultas ao banco foram parametrizadas por meio de *prepared statements*.

6 CONSIDERAÇÕES FINAIS

A segurança dos dados armazenados por uma aplicação é assunto de grande importância para pessoas e organizações. Perder informações ou ter dados adulterados pela ação de usuários mal-intencionados pode significar prejuízo financeiro, perda de recursos ou tempo investidos, entre outros inconvenientes.

O presente estudo teve por objetivo levar a uma melhor compreensão a respeito dos ataques por injeção de SQL, bem como determinar a eficácia do uso da API PDO, com *prepared statements* e parametrização, na proteção de sistemas contra essa forma de ataque.

Nos testes realizados, por meio de injeções manuais em um campo de formulário ou com o uso de uma ferramenta especializada, a PDO mostrou-se 100% segura contra ataques por injeção de SQL, desde que, é claro, utilizada corretamente. Os dados entrados pelo usuário no sistema devem ser devidamente parametrizados para que, por meio de *prepared statements*, sejam enviados ao SGBD em momento posterior à preparação da consulta, impedindo que comandos injetados sejam interpretados como código a ser executado.

O uso da PDO sem *prepared statements* e parametrização das consultas resultou em um sistema de *login* vulnerável, como de fato era esperado. São os *prepared statements* que, ao impedir a execução dos parâmetros como parte das instruções SQL, efetivamente impedem a execução de injeções. Vale ressaltar que é necessário fazer uso desta estratégia mesmo que os parâmetros não venham diretamente de um formulário. Se, por exemplo, consultas são montadas no sistema a partir de informações presentes no banco de dados que haviam sido entradas pelo usuário em um momento anterior, injeções de segunda ordem (comandos injetados pelo usuário que não foram executados em um primeiro momento, mas foram armazenados no banco) poderiam ocorrer se tais consultas não fossem parametrizadas.

A utilização dessa estratégia de segurança contra injeções de SQL adquire importância ainda maior na medida em que métodos de validação negativa – o escape de caracteres considerados perigosos – acabam por eliminar, por vezes, caracteres legítimos (por exemplo, a apóstrofe de nomes americanos como O'Brian). Para além disso, é difícil prever todos os tipos de entradas de dados que poderão causar problemas em tentativas de ataques.

Tendo em vista que o recurso de *prepared statements* e parametrização encontram-se presentes na linguagem PHP, por meio de uma API (PDO) orientada a objetos e com suporte a vários tipos de SGBD, não há razão para não fazer uso dessa camada de proteção. Cabe ressaltar que a PDO é parte das especificações da própria linguagem PHP e, assim, encontra-se presente em sua instalação padrão.

Apesar disso, é preciso atentar para o fato de que a pesquisa abordou apenas uma forma de injeção de código que, por sua vez, caracteriza-se como uma das categorias de ataques existentes. Portanto, ainda que *prepared statements* consigam defender um sistema adequadamente contra injeções de SQL, outras formas de proteção permanecem sendo necessárias para que seja possível prevenir diferentes formas de ataque e dirimir vulnerabilidades.

Tornar sistemas informatizados mais seguros implica planejamento e testes cuidadosos. Felizmente, ferramentas como a PDO e técnicas como *prepared statements* auxiliam nessa tarefa tão importante e difícil.

REFERÊNCIAS

20 THINGS I learned about browsers and the web. Disponível em: <http://www.20thingsilearned.com/en-US/home>. Acesso em: 26 mar. 2014.

ARCHITECTURE of the World Wide Web, Volume One: W3C Recommendation 15 December 2004. Disponível em: <http://www.w3.org/TR/webarch/>. Acesso em: 20 mar. 2014.

ACHOUR, Mehdi et al. *PHP manual*. 2014. Disponível em: <http://www.php.net/manual/en/>.

CATEGORY: vulnerability. Disponível em: <https://www.owasp.org/index.php/Category:Vulnerability>. Acesso em: 11 abr. 2014.

CHURCHER, Clare. *Beginning database design:* from novice to professional. New York: Apress, 2007.

CODE Injection. 2013. Disponível em: <https://www.owasp.org/index.php/Code_Injection>. Acesso em: 1 mai. 2014.

CONNECTIONS. Disponível em: <http://php.net/manual/en/mysqli.quickstart.connections.php>. Acesso em: 19 agosto 2014.

CONNECTIONS AND connection management. Disponível em: <http://br1.php.net/manual/en/pdo.connections.php>. Acesso em: 19 agosto 2014.

CREATE function. Disponível em: <https://mariadb.com/kb/en/create-function/>. Acesso em: 4 mai. 2014.

DAMAS, Luís. *SQL*: structured query language. 6 ed. Rio de Janeiro: LTC, 2007.

DATE, Christopher J. *Introdução a sistemas de banco de dados*. 8. ed. Rio de Janeiro: Elsevier, 2003.

DEITEL, Paul; DEITEL, Harvey. *Java:* como programar. 8 ed. São Paulo: Pearson, 2010.

ESCAPING and SQL injection. Disponível em: <http://docs.oracle.com/cd/E17952_01/apis-php-en/apis-php-mysqli.quickstart.prepared-statements.html>. Acesso em: 15 jun. 2014.

FREATURES. Disponível em: <https://github.com/sqlmapproject/sqlmap/wiki/Features>. Acesso em: 29 nov 2014.

GILMORE, W. Jason. *PHP and MySQL:* from novice to professional. 4 ed. New York: Apress, 2010.

HALFOND, William G. J; VIEGAS, Jeremy; ORSO, Alessandro. *A classification of SQL injection attacks and countermeasures*. Artigo – Georgia Institute of Technology, 2006.

HEUSER, Carlos Alberto. *Projeto de banco de dados*. 6. ed. Porto Alegre: Bookman, 2009.

HISTORY of the Web. Disponível em: <http://www.webfoundation.org/about/vision/history-of-the-web/>. Acesso em: 25 mar. 2014.

HTML5. Disponível em: <http://www.w3.org/TR/html5/>. Acesso em: 26 mar. 2014.

INTRODUCTION. Disponível em: <http://www.php.net/manual/en/intro.pdo.php>. Acesso em: 03 abr. 2014.

MARIADB. Disponível em: <http://en.wikipedia.org/wiki/MariaDB>. Acesso em: 10 abr. 2014.

MARIADB IS a binary drop in replacement for MySQL. Disponível em: <https://mariadb.com/kb/en/mariadb-versus-mysql-compatibility/>. Acesso em: 10 abr. 2014.

MEUCCI, Matteo; Muller, Andrew (Orgs.). *OWASP testing guide 4.0*. Disponível em: <https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents>. Acesso em: 25 out. 2010.

MORRISON, Michael. *Head First JavaScript*. Sebastopol: O'Reilly, 2008.

MySQL CUSTOMERS. Disponível em: <http://www.mysql.com/customers/>. Acesso em: 10 abr. 2014.

NAMING and addressing: URIs, URLs, ….
Disponível em: <http://www.w3.org/Addressing/>. Acesso em: 28 mar. 2014.

OVERVIEW. Disponível em: <http://br1.php.net/manual/en/mysqli.overview.php>. Acesso em: 03 abr. 2014.

PREPARED statements and stored procedures. Disponível em: <http://www.php.net/manual/pt_BR/pdo.prepared-statements.php>. Acesso em: 15 jun. 2014.

SQL injection. 2014. Disponível em: <https://www.owasp.org/index.php/SQL_Injection>. Acesso em: 1 mai. 2014.

SNYDER, Chris; MYER, Thomas; SOUTHWELL, Michael. *Pro PHP security:* from application security principles to the implementation of XSS defenses. 2 ed. New York: Apress, 2010.

STORED procedure overview. Disponível em: <https://mariadb.com/kb/en/stored-procedure-overview/#why-use-stored-procedures> Acesso em: 11 mai. 2014.

SU, Zhendong; WASSERMANN, Gary. *The essence of command injection attacks in web applications*. Artigo – University of California, 2006.

TECHNIQUES. Disponível em: <https://github.com/sqlmapproject/sqlmap/wiki/Techniques>. Acesso em: 7 jun. 2014.

TESTING for SQL injection (OWASP-DV-005). Disponível em: <https://www.owasp.org/index.php/Testing_for_SQL_Injection_%28OWASP-DV-005%29>. Acesso em: 12 mai. 2014.

USAGE. Disponível em: <https://github.com/sqlmapproject/sqlmap/wiki/Usage>. Acesso em: 17 out. 2014.

WILLIAMS, Robin; TOLLETT, John. *Web design para não-designers*. Rio de Janeiro: Editora Ciência Moderna Ltda., 2001.

WYKE-SMITH, Charles. *Stylin' with CSS:* a designer's guide. 3 ed. San Francisco: New Riders, 2013.

ZANDSTRA, Matt. *PHP objects, patterns, and practice*. 3 ed. New York: Apress, 2010.