SQLi.md

SQL Injection

Conceito

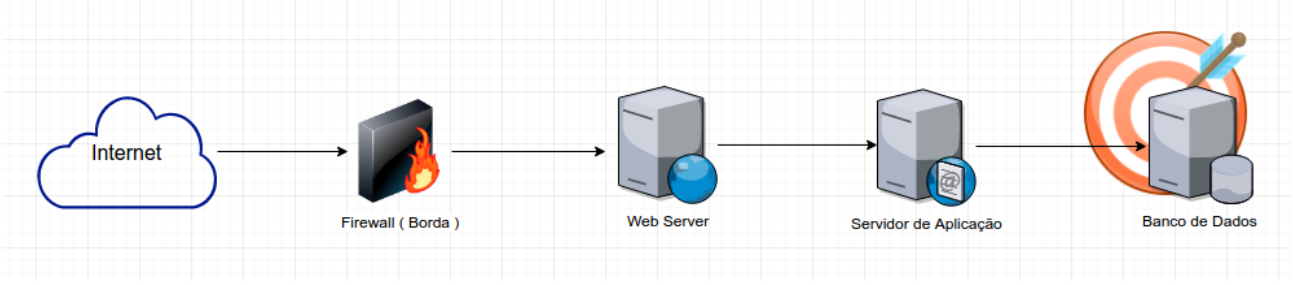
As falhas baseadas em Injeção de código são aplicaveis a qualquer base de dados como injeção de código SQL, ou inserção de dados em uma base LDAP, ocorrem quando dados não confiavéis são enviados para um interpretador como parte de um comando ou consulta. Os dados manipulados pelo atacante podem iludir o interpretador para que este execute comandos indesejados ou permita o acesso a dados não autorizados, *** trata-se de como fazer a pergunta errada na hora certa ou melhor na inesperada*** e explorar as respostas obtidas.

- Segundo a publicação "SQL Injection Defenses" de Martin Nystrom um Ataque baseado em SQLi ocorrido em 2004 em uma empresa chamada CardSystems Incorporated cujo core bussiness baseia-se em meios de pagamentos levou ao roubo de 263,000 números de cartões de créditos de uma base de dados, mais do que isso, a base de dados em questão ***Não utilizava criptografia*** para armazenamento de dados dos cartões de crédito em questão sendo este conhecido como um dos maiores danos causados por cyber ataques da história.

> **COMMAND INJECTION ATTACKS CAN PUT YOU OUT OF BUSINESS**
>
> SQL injection and other types of command injection attacks can ruin entire businesses. For example, an SQL injection attack was revealed in June 2005 in which a credit card payment processing company called CardSystems had 263,000 credit card numbers stolen from its database. Even worse, since the credit card numbers were stored in its database in an unencrypted form, over 40 million credit card numbers were potentially exposed to the attack! The attack was arguably the worst cyber-attack of all time at the time of writing this book, and was investigated by Congress and the FTC. CardSystems lost large amounts of business and its assets were acquired by another company.
>
> In addition, awareness of SQL injection vulnerabilities seems to be on the rise. In the first half of 2004, there were 57 SQL injection vulnerabilities reported to the BugTraq security vulnerability mailing list (`www.securityfocus.com/archive/1`), and that number more than tripled to 194 during the first half of 2005 (Ng 2006). In this chapter, we show how SQL injection attacks work and discuss how they can be prevented.

Como esses ataques se relacionam com Aplicações Web?

Se considerarmos a maioria das aplicações voltadas para internet a arquitetura utilizada será semelhante ao modelo descrito abaixo:



Com base nessa arquitetura saindo do Browser do usuário todos os dados processados serão enviados e recebidos utilizando HTTP ou HTTPS, o servidor de web por usa vez recebe e faz o "parse" desses dados e envia as informações relevantes para o servidor de aplicação atuando como um proxy reverso, uma das vertentes do SQLi é exatamente explroar essas requisições principalmente se estivermos falando de HTTP que trafega tudo texto, o chamado **plain-text encoding** o que permite que essa informação e as consultas direcionadas a base de dados sejam interceptadas e alteradas.

Para cada campo em que uma aplicação recebe requisições e manipula dados temos uma oportunidade para explorar SQLi, nesse modelo de abordagem o atacante tantará manipular as requisições e a partir de análise de retornos dicernir campos onde dados poderão ser inseridos, para entender esse processo de inserção vale a pena analiar como as consultas são feitas usando a linguagem SQL:

Querys utilizando SQL geralmente são executadas usando uma biblioteca especifica a linguagem que você está usando, PHP, Java, C# etc, Essa biblioteca fornece instruções estruturadas que permitem construir as tais Querys SQL e formatalas corretamente para que sejam compreendidas pelo banco de dados.

Passo a passo de uma Query SQL:

A maioria das bibliotecas SQL compartilha um conjunto comum de objetos usados para manipular dados SQL:

- ***Statement:*** Qualquer instrução "padrão" da linguagem SQL, como insert, update, exclude, truncate, drop, e assim por diante.
- ***Prepared Statements:*** Instrução SQL declarada que permanece na memória com variáveis vinculadas a ela.
- ***Callable Declared :*** Uma instrução construída e armazenada inteiramente no banco de dados como um procedimento, os famosos procedures que podem ser chamadas referenciando o nome dado a instrução no banco de dados.

Ao executar uma query no banco de dados, geralmente a API utilizada para consulta percorrera estas etapas:

1. Conectar-se ao banco de dados com um nome de usuário e uma senha armazenados;
2. Criação de uma instrução SQL com um dos conjuntos citados acima;
3. Executar a instrução com a query de consulta relacionada a linguagem. Os dados serão recolhidos e mantidos para em seguida serem entregues como retorno a query executada;
4. Verificação do código de retorno para determinar se os resultados obtidos são válidos e se a consulta fora executada normalmente. Se o retorno da consulta for com êxito, o programa irá estruturar os resultados (linhas), e estruturar os campos (colunas) e, em seguida, processar os dados.
5. Desconectar-se do banco de dados;

Em casos onde a operação a ser executada resulta em adicionar conteúdo a uma base de dados o Desenvolvedor provavelmente utilizará a API de sua aplicação e deverá percorrer os seguintes passos:

1. Conectar a base de dados;
2. Construir uma instrução para update;
3. Executar a instrução de update utilizando um dos verbos citados a pouco, insert, update ou delete;
4. Verificar o retorno da operação (Geralmente trata-se de um valor íntegro contendo o número de linhas afetadas pela instrução);
5. Executar um commit da transação salvando permanentemente o estado recebido;
6. Desconectar da base de dados;

Como as aplicações de web recebem esses dados?

Eis aqui o ponto mais importante para esta aula, em geral para que sua aplicação receba esses dados geralmente eles deverão estar encapsulados usando o protocolo HTTP sendo que a informação recebida poderá ser estruturada usando XML, HTML ou JSON. O esquema mais simples para armazenamento desses dados é utilizando o bom é velho formato " chave=valor " com por exemplo ID=342424.

A questão a ser considerada é que independente do formato no HTTP 1.1 esses dados são processados em texto puro, mesmo que esteja sendo utilizado criptografia TLS, o dado estará criptografado mas ainda acessível ao cliente de onde será originará a conexão e talvez o ataque usando SQLi;

> Alguns "servlets" e plugins específicos vinculados ao browser utilizam outros protocolos para processamento de informação, Na linguagem Java por exemplo costumava-se utilizar RMI para comunicação com o servidor, quanto mais específico for o formato mais complicado tende a ser a exploração usando SQLi pois existe um tratamento maior no dado sendo enviado ao banco.

O exemplo abaixo refere-se a uma requisição tradicional em Java utilizando como base o modelo "chave=valor" para criação da requisição com protocolo HTTP:

```
{      public void doGet
              (HttpServletRequest req,
               HttpServletResponse res)
       throws ServletException, IOException
       {String name = req.getParameter("name");
```

Nesse exemplo temos uma requisição do tipo HttpServletRequest com o melhod "getParameter" que permitirá a manipulação de strings pelo nome o mesmo acontece na maioria das linguagens de programação.

A sacada sobre esse formato é que a informação dentro da chave/valor pode ser manipulara caso não existam processos de validação, daí a primeira oportunidade para nossos trabalhos com SQLi

---

Motivações e Ataques:

O atacante tende a sempre procurar por Vulnerabilidades no código pois essa será provavelmente sua melhor chance de chegar a base de dados ou seja, a informação alvo, isso ocorre porque por questões de boas práticas e padronizações de segurança é comum que o banco de dados esteja alocado em redes isoladas e mais confiáveis que as redes voltadas a aplicações de frontend. Isso torna sua aplicação uma boa opção para porta de entrada.

> Tenha em mente que em geral ataques de engenharia social são um meio muito mais eficaz de colher dados de bancos de dados. Um atacante ganha acesso ao sistema a partir de alguém do interior da empresa (um sysadmin ou developer por exemplo) a partir desse acesso e talvez de uma escalada de privilégios o atacante poderá puxar informações do banco de dados para fora da empresa, aplicar técnicas de ***Ransomware*** e/ou enviar esses dados para o exterior.

O que possibilita esse método de ataque?

Esse tipo de Vulnerabilidade se torna real a partir de aplicações que permitam ao atacante enviar comandos diretamente ao banco de dados sem que esses comandos ou Querys sejam interpretadas pelo servidor de aplicação ou validadas mediante o próprio código da aplicação.

Quais são os exploits mais comuns relacionados?

- Programas que criam instruções, substituindo a entrada do usuário em qualquer parte da instrução;
- Entradas que são passadas para o mecanismo SQL sem a filtragem adequada;

> Outro ponto a se considerar é o excesso de informações, Normalmente, mensagens de erro detalhadas são passadas de volta para o usuário a depender da mensagem isso permite que um usuário mal-intencionado saiba se seus ataques de injeção são bem-sucedidos ou não.

Etapas envolvidas em processo de SQLi

As etapas que envolvem processos de ataques baseados em SQL injection são basicamente as mesmas descritas no Ciclo de Vida de uma VA, sendo que três delas acabam recebendo maior foco:

1. ***Reconhecimento ("Getting Information" ou Levantamento de Informações)***

A partir do momento que um alvo é estabelecido o atacante deverá começar o processo de avaliação e levantamento de informações sobre o ambiente de seu alvo você, ele vai começar a determinar a paisagem de seu alvo. Para fazer isso, ele provavelmente usará ferramentas como o nmap para encontrar hosts que respondem no ambiente. Ele também procurará serviços executados nesses hosts e versões de serviços nesse ambiente. Normalmente, será um ataque mais eficaz se ele poder encontrar um servidor web que aparentemente possui conexão com alguma base de dados, formulários e campos de preenchimento são um ótimo indício da existência dessas conexões.

1. ***Levantamento de Vulnerabilidades***

Uma vez que ele descubra o que está sendo executado em seu ambiente, o invasor irá procurar por vulnerabilidades que ele possa atacar. No caso de SQLi isso envolve testes lógicos e análise de comportamento de formulários e campis de preenchimento, o que pode ser um processo bem longo e massivo, mas para fazer isso, ele pode e provavelmente irá usar ferramentas como Nessus, SQLMap, SQLNinja ou Arachni (Essa última é mais um gosto pessoal do Professor). Essas ferramentas possuem recursos específicos para procurar por "pontos fracos" em Instruções SQL. O Nessus (veja a Figura abaixo) por exemplo possui um perfil específico de busca para este tipo de informação, testaremos algumas dessas ferramentas nas próximas aulas.

1. ***Explorando o Exploit encontrado***

Nessa etapa o atacante já terá uma boa noção de por onde agir e potenciais Vulnerabilidades, nesse caso ignorei a parte referente a análise de falso positivos e estou indo direto a ofensiva, o atacante começará provavelmente tentando "confundir" a aplicação (Em inglês utilizamos o termo fuzzig para isso)fazendo consultas inesperadas e principalmente procurando por mensagens de erro detalhadas que possam elucidar a estrutura SQL até então desconhecida.

A medida que o invasor entende a estrutura das instruções SQL, ele tenta injetar código SQL nos campos de entrada, observando as indicações de que seu ataque está sendo bem sucedido através dos retornos obtidos via interface do usuário.

Tipos de Ataques

Existem basicamente três classificações para tipos de ataques contra baseados em SQL:

- Ataques de visualização completa (Full View);
- Ataques de "Blind SQL Injection";
- Ataques automatizados de de SQL Injection;

Full-view SQL injection attacks:

Esse é o tipo tradicional de ataque, baseado quase que inteiramente em mensagens de erro detalhadas e dados não filtrados retornados pela aplicação e exibidos na interface do usuário.

A informação obtida nesses processo ajuda o atacante a elaborar e refinar seu ataque. Ele provavelmente começará pelo processo de "fuzzing" executando requisições e testes nos campos de preenchimento dentro da aplicação com o objetivo de provocar erros e analisar seus resultados. Com cada erro, ele estará aprendendo sobre como os comandos SQL são estruturados, a linguagem de programação da aplicação e a linguagem da própria base de dados.

O atacante começará injetando comandos nos parâmetros de entrada, com a finalidade de alterar a cláusula ***"where"*** para retornar um conjunto de resultados maior do que o que o esperado pelo desenvolvedor.

> Esse geralmente é o primeiro mais simples método de ataque pois é muito mais fácil do que alterar as colunas retornadas ou as tabelas consultadas.

Blind SQL Injection:

Processos de "Blind SQLi" são aqueles onde sua aplicação ou alvo possui certo nível de tratamento referente aos retornos exibidos ao executar consulta, nesses casos um consulta incorreta ou usando caracteres inesperados por exemplo retornaria uma página de erro genérica ou simplesmente não retornaria nada. Ou seja sua ofensiva ou testes deverá proceder sem ter como base ostentarmos e feedbacks do banco que ajudariam na hora de refinar as próximas consultas.

> Depois que o SQL Injection se tornou conhecido desenvolvedores passaram a ser advertidos e se preocuparem com a necessidade de **desativar o formato verbose é reduzir as mensagens de erro a serem devolvidas aos usuários*** Ainda assim os atacantes começaram a se habituar com esse novo formato e continuaram evoluindo na exploração de SQL uma vez que com ou sem liga de retorno às querys ainda são executadas no banco de dados.

Neste caso como identificar um exploit para SQLi?

Para explorar Vulnerabilidades baseadas em Blind SQLi o primeiro passo é identificar um exploit válido, existem algumas abordagens a serem usadas para obter essa informação:

1. Forçar mensagens de erro genéricas:

Mesmo que as aplicações substituam as mensagens de erro do Banco por mensagens genéricas ainda assim estamos falando de mensagens de erro, ou seja, sua simples existência é o suficiente para o atacante inferir que está diante de um possível exploit para SQLi.

> Como um exemplo bem simples considere o uso que fizemos de uma "quote" nos testes anteriores, mesmo que ao inserir esse símbolo uma exceção não seja exibida o fato de obter uma mensagem de erro que não apareceu em outras consultas já sugere que o a aplicação pode não estar conseguindo tratar esse caractere especial em específico.

2. Efeitos colaterais demonstrados diante de algumas consultas:

Outro método comum para Blind SQLi é apostar na criação de Querys que criem efeitos colaterais no Banco de Dados, a técnica mais antiga é conhecida para isso é o uso dos chamados "Time Based Attacks" nesses casos inferimos se uma Query corresponde a um exploit ou não observando a diferença de tempos entre uma consulta bem sucedida e uma falha de execução, outra questão relativa a essa técnica é que DBMS específicos possuem funções específicas que podem gerar atrasos no tempo de resposta.

Por exemplo, no MySQL server da Microsoft é possível gerar uma pausa de 5 segundos usando a seguinte instrução:

```
WAITFOR DELAY ‘0:0:5’
```

De forma similar em algumas versões do MySQL server podemos utilizar a função abaixo e obter o mesmo resultado:

```
SLEEP(5)
SELECT 1 FROM t1 WHERE SLEEP(1000);   # Exemplo básico de select com timeout de 1000 Segundos;
```

Funções como essa existem pela necessidade de estabelecer timeout sob determinadas execuções, o que não impede um atacante de utiliza-las para determinar qual o DBMS em uso, por isso esse tipo específico de consulta em un processo de SQLi recebe o nome de "Time Based Attack".

Ainda falando sobre processos classificados como "Blind SQL Injection" um outro ponto importante é simplesmente observer a resposta obtida com testes de strings simples que já executamos anteriormente, por exemplo:

Considere que mesmo sem retornos de output ainda podemos ter um formulário onde o valor de entrada não está sendo tratado antes de se tornar a query a ser executada no banco, neste caso uma abordagem para confirmar essa informação poderia ocorrer dessa forma:

Primeiro gere uma query que sempre terá retorno falso:

```
‘ AND ‘1’=’2
```

Depois uma query com retorno sempre verdadeiro:

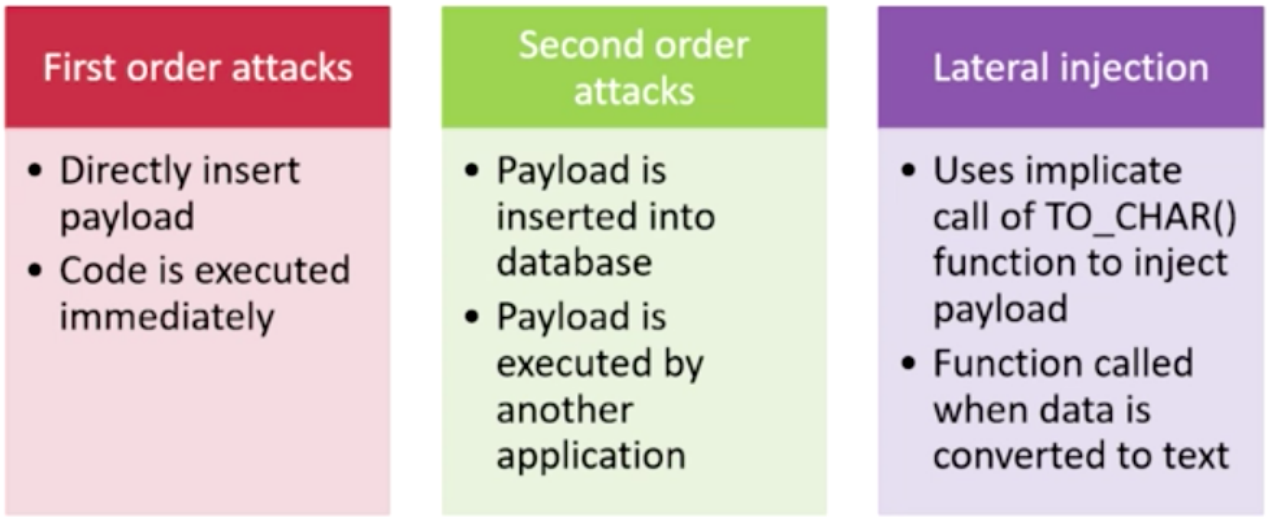
```
‘ OR ‘1’=’1
```

Se o tratamento das consultas anteriores estiver sendo executado a nível de aplicação é provável que em ambos os casos tenhamos o mesmo retorno uma vez que a consulta seria filtrada e provavelmente está sendo depurada antes de chegar ao banco.

> O exemplo acima trata de um método onde o resultado é interpretativo ou seja, presumimos que a partir do resultado temos uma determinada situação. Esta é a questão sobre Blind SQLi, os resultados tem que ser interpretados e muitas vezes o atacante tem de considerar hipóteses para seguir em determinadas linhas de ações em seu ataque.

Tipos de Inserts e manipulações:

É muito comum que seja utilizada uma segunda classificação baseada no tipo de inserção e manipulação feita no banco:



Defesa

Com relação a prevenção e mitigação de riscos relacionados a SQLi existem certas abordagens que podem ser utilizadas, sendo algumas delas ativas, baseadas em alteração e cuidados com código, como uso de conectores para bancos, validação de campos e de Querys antes de seu disparo ao DBMS (Database Management System), e outros são reativos baseados na configuração do próprio DBMS, bloqueio de ataques em tempo real e análise de Querys em execução.

Código Seguro

Se considerar a natureza da maioria dos ataques utilizados até então, o código fonte da aplicação passa a ser com certeza o ponto mais importante de sua defesa, ou seja, a ideia é evitar que código vulnerável seja entregue em produção, faça isso usando instruções pré-configuradas de conexão, os chamados ***prepared statements*** e também usando filtragem de entrada para suas instruções SQL.

Exemplo usando PHP

O exemplo abaixo demonstra uma instrução de conexão baseada em PHP, Nesta instrução estamos tomando o parâmetro **$name** como entrada de usuário e, em seguida, usando esse parâmetro diretamente em uma consulta sem filtragem. Isso permitiria que um invasor injetasse comandos SQL no parâmetro **$name** conforme exemplos anteriores usados em aula.

```
// Recuperando o parâmetro $name de uma HTTP request...
$name = $_REQUEST("name");
// Configurando a Quey para busca por um suposto campo SSN...
$query = "select ssn from customers where name = '" . $name . "'";
// Execução da Query...
$query->execute();
// Recuperando os dados de nossa SSN como resultado da query...
$ssn->$query->fetchAll();
```

Para reparar o modelo anterior neste exemplo, usaremos uma "prepared statement", Essas chamadas instruções preparadas foram criadas a partir do PHP5, utilizando como base PHP Data Object (PDO), a idéia do PDO é preparar os parâmetros antes de uma query para só então vincular a informação recebida na Requisição HTTP ao driver do DBMS (Neste exemplo MYSQL):

```
// Recuperando o parâmetro $name de uma HTTP request...
$name = $_REQUEST("name");
// Configurando a Query para busca por um suposto campo SSN...
// Neste caso utilizando uma Instrução Preparada ( Prepared Statement ) e fazendo bind no parâmetro name:
$query = $dbhandle->prepare("select ssn from customers where name = ?")
// Agora executamos a consulta, substituindo a variável "$name" usando
// um array para alimentar a API:
$query->execute($query,array($name));
// Recuperando os dados de nossa SSN como resultado da query...
$ssn->$query->fetchAll();
```

Uma boa base para estudo e entendimento dessa estrutura é a documentação oficial do php.net, disponível em: http://php.net/manual/pt_BR/security.database.sql-injection.php

O w3schools possui ótimos exemplos de como utilizar Prepared Statements no PHP: https://www.w3schools.com/php/php_mysql_prepared_statements.asp

> É quase sempre uma boa dica procurar portais oficiais ou paginas de manual relativas a linguagem de programação que está se utilizando, por ter se popularizado muito hoje não faltam informações sobre prevenção contra SQLi para a grande maioria das linguagens de programação para Frontend.

Referências Úteis:

Conhecendo os conceitos de instruções SQLi os processos de mitigação serão quase sempre os mesmos, variando apenas na maneira como são aplicados em cada Linguagem, no caso de PHP vale a pena começar pelo seguinte:

1. Uso de Prepared Statements:

Conforme citado anteriormente a partir da versão 5 o PHP passou a oferecer uma relação de intruções pré configuradas para comunicação com databases tanto no uso de conectores Mysqli como PDO;

O link da w3schools é uma boa fonte para começar a entender este assunto: Prepared Statements and Bound Parameters

2. Filtro e validação de dados antes dos processos de conexão com o Banco:

Muitos ataques (Na verdade a maioria deles) são baseados na inserção de valores inesperados na query causando excessões nas respostas, é o caso do uso do "'".

Uma opção para mitigar esse risco é aplicação de filtros para validação desses dados, segue o exemplo do w3schools e do PHP.net com base documentada sobre os principais filtros para PHP5:

- PHP 5 Filter Functions
- Sanitize filters

3. "Controlar" os valores que aparecem na URI:

Campos como id e valores referentes a campos de tabelas do Database devem sempre que possível ser obfuscados na URL criada para conexão, esse processo dificulta (e muito) processos automatizados de SQLi.

Nesse caso algumas soluções são aplicaveis como por exemplo:

- Uso de funções para obscurecer valores;
- Uso de tokens;
- Criptografia de campos (Método controverso);

Exemplo usando Javascript:

Considere mais um exemplo, este baseado em JavaScript:

No exemplo abaixo os parâmetros digitados pelo usuário são automaticamente armazenados como usuário e senha nos parâmetros "user" e "pass", em seguida estes parâmetros são convertidos em uma instrução ao database, nesse caso para cadastro de dados:

```
String user=request.getParameter("user");
String pass=request.getParameter("pass");
String sql = "Insert into accounts (username.password) values (?,?)";
```

Conforme discutido durante a apresentação deste conteúdo, uma vez que não exista tratamento da entrada de dados no formulário quaisquer valores são aceitáveis, logo uma consulta inteira pode ser inserida ou uma regra booleana (os chamados boolean-based) para validação baseada em OR por exemplo.

Para melhorar o exemplo anterior adicionaremos uma verificação básica baseada no uso de ***Prepared Statements***. Hoje em dia a maioria das linguagens de programação possuem modelos de código pré moldados para comunicação com o banco de dados, No caso do JavaScript com o uso de PreparedStatement conseguiremos isolar os parâmetros que o usuário passa nos campos "user" e "pass", conforme exemplo abaixo:

```
PreparedStatement stmt = connection.prepareStatement(sql);
stmt.setString(1,user);
stmt.setString(2,pass);
stmt.execute();
```

Material de Referência:

o Livro "SQL INjection Defenses" de Martin G. Nystrom Publicado em 2007 é ainda hoje a melhor referência teórica sobre motivações e estrutura utilizada em Ataques do tipo SQLi:

- SQL Injection Defenses, Trecho gratuito do livro no Google Books

Conforme descrito no exempo dado no tópico sobre Defesa, existe uma boa base para tratameto de Querys principalmente relativo ao uso de PDO na documentação da página php.net:

- php.net: Injeção de SQL

Free Software, Hell Yeah!