SQL Injection (SQLi): Entendendo e identificando a vulnerabilidade em aplicações

Quase todas as aplicações web utilizam um banco de dados para armazenar os vários tipos de informações de que necessita para operar. Por exemplo, uma aplicação web implantado por um varejista on-line pode usar um banco de dados para armazenar as seguintes informações:

- As contas de usuário, credenciais e informações pessoais;
- As descrições e preços dos itens para venda;
- Ordens de serviços/compras, extratos de conta e detalhes de pagamento;
- Os privilégios de cada usuário dentro da aplicação.

Os meios de acesso à informação no banco de dados é Structured Query Language (SQL). SQL pode ser usado para ler, atualizar, adicionar e eliminar informações contidas no banco de dados.

SQL é uma linguagem interpretada, e aplicações web comumente constroem instruções SQL que incorporam dados fornecidos pelo usuário. Se isso for feito de maneira insegura, o aplicativo pode ser vulnerável a injeção de SQL. Este é uma das vulnerabilidades mais notórias para afetar aplicações web. Nos casos mais graves, a injeção de SQL pode permitir que um invasor anônimo ler e modificar todos os dados armazenados no banco de dados, e até mesmo assumir o controle total do servidor no qual o banco de dados está em execução.

Enquanto a consciência de segurança de aplicações web evoluiu, as vulnerabilidades de injeção SQL tornaram-se gradualmente menos difundida e mais difíceis para detectar e explorar. Muitas aplicações modernas evitam a injeção SQL através do emprego de APIs que, se usados corretamente, são segura contra

ataques de injeção SQL. Nestas circunstâncias, a injeção de SQL tipicamente ocorre em casos pontuais, onde estes mecanismos de defesa não podem ser aplicados. Encontrando injeção de SQL é algumas vezes uma tarefa difícil, exigindo perseverança para localizar um ou dois casos em uma aplicação onde não tenham sido aplicados os controles habituais.

Como esta tendência tem se desenvolvido, métodos para encontrando e explorar falhas de injeção SQL têm evoluído, utilizando indicadores mais sutis de vulnerabilidades e técnicas de exploração mais refinadas e poderosas. Vamos começar por analisar os casos mais básicos e, em seguida, ir para descrever as mais recentes técnicas para a detecção e exploração as cegas.

Uma vasta gama de bancos de dados são utilizados para suportar aplicações web. Embora os fundamentos de injeção SQL são comuns à grande maioria destes, existem muitas diferenças. Esta extensão de pequenas variações na sintaxe até divergências significantes no comportamento e funcionalidade que pode podem afetar os tipos de ataques que você poderá buscar. Vamos restringir nossos exemplos para os três bancos de dados mais comuns que são susceptíveis de encontrar – Oracle, MS-SQL e MySQL. Sempre que aplicável, vamos chamar a atenção para as diferenças entre estas três plataformas. Equipado com as técnicas descritas aqui, você deve ser capaz de identificar e explorar vulnerabilidades de injeção de SQL contra qualquer outro banco de dados através da realização de algumas pesquisas adicionais rápidas.

Explorando uma vulnerabilidade básica

Considere uma aplicação web desenvolvida por um varejista de livros que permite os usuários pesquisarem por produtos pelo nome do autor, título, editora, e assim por diante. O catálogo

inteiro de livros está em um banco de dados e a aplicação usa consultas SQL para obter detalhes sobre os diferentes livros baseados nos termos da pesquisa informados pelos usuários.

Quando um usuário pesquisa por todos os livros de uma editora chamada **XPTO**, a aplicação realiza a seguinte consulta:

```
SELECT autor,titulo,ano FROM livros WHERE editora = 'XPTO' AND publicado=1
```

Esta consulta faz uma verificação em cada linha do banco de dados dentro da tabela de livros, extrai cada um dos registros onde a editora seja **XPTO** e o campo publicado esteja com valor 1, e retorna o conjunto de registros encontrados. A aplicação então processa este conjunto e apresenta para o usuário dentro do HTML da página. Nesta consulta, as palavras do lado esquerdo do sinal de igual são palavras-chaves do SQL e o nome da tabela e das colunas dentro do banco de dados. Esta parte da consulta foi construída pelo programador quando a aplicação foi criada. A expressão XPTO foi enviada pelo usuário. String de dados em consultas SQL devem ser encapsuladas dentro de aspas simples (‘) para separar isto do resto da consulta.

Agora, considere o que poderia acontecer se um usuário buscar por todos os livros publicados pela **O’Reilly**. A aplicação realizará a seguinte consulta:

```
SELECT autor,titulo,ano FROM livros WHERE editora = 'O'Reilly' AND publicado=1
```

Neste caso, o interpretador da consulta irá executar a string de dados da mesma forma que antes. Vai passar o dado, o qual foi encapsulado dentro das aspas simples, e obter o valor **O**. E então encontra a expressão **Reilley’**, a qual não é uma sintaxe SQL válida, e causará um erro:

```
Incorrect syntax near ‘Reilly’.
Server: Msg 105, Level 15, State 1, Line 1
Unclosed quotation mark before the character string ‘
```

Quando a aplicação se comporta desta forma, ela é vulnerável a injeção de SQL. Um atacantes pode fazer o input de dados usando uma aspa simples para terminar a string que ele controla. Então ele pode escrever um SQL arbitrário para modificar a consulta que o programador preparou para que a aplicação executasse. Nesta situação, por exemplo, o atacante pode modificar a consulta para retornar todos os livros do catálogo da loja virtual inserindo o seguinte termo na busca:

```
XPTO' OR 1=1--
```

Isto fará com que a aplicação realizar a seguinte consulta:

```
SELECT autor,titulo,ano FROM livros WHERE editora = 'XPTO' OR 1=1--' AND publicado=1
```

Isto modifica a cláusula WHERE da consulta do programador para adicionar uma segunda condição. O banco de dados verificar cada linha na tabela de livros e extrai cada registro da coluna editora que tenha o valor XPTO ou onde 1 for igual a 1. Como sempre 1 será igual a 1, o banco de dados retornará cada um dos registros na tabela de livros.

O duplo hífen no input do atacantes é uma expressão SQL para dizer que ao interpretador da consulta que o resto da linha é um comentário e deve ser ignorado. Este truque é extremamente útil em alguns ataques SQL, porque permite você ignorar o restante da consulta criada pelo programador para a aplicação. Neste exemplo, a aplicação encapsula a string enviada pelo usuário com a aspa simples. Como o atacante terminou a string que ele controla e injetou um SQL adicional, ele precisa manipular a aspa da direita para evitar erro de sintaxe, como no exemplo do O'Reilly. Ele consegue fazer isto adicionando um duplo hífen, fazendo com que o restante da consulta seja tratada como um comentário. Em MySQL, você precisa incluir um espaço depois do duplo hífen ou usar um caractere hash para especificar um comentário.

A consulta original também tem o acesso controlado somente

para os livros publicados, porque ele restringe com **AND publicado=1**. Ao injetar a sequência de comentário, o atacante ganhou acesso não autorizado retornando os detalhes de todos os livros, publicados ou qualquer outro tipo.

Em alguns casos, uma alternativa para manipular as aspas sem usar o símbolo de comentário é “balanceando as aspas”. Você finaliza o input injetado uma aspa na string de dados que requer uma aspa do lado direito para encapsular. Por exemplo, com o termo de busca:

Wiley' OR 'a' = 'a

Resulta na seguinte consulta:

SELECT autor,titulo,ano, FROM livros WHERE editora = 'XPTO' OR 'a'='a' and published=1

Isto é perfeitamente válido e atinge o mesmo resultado que o ataque **1 = 1** para retornar todos os livros publicados pela XPTO, independentemente de serem publicados.

Este exemplo mostra como a lógica da aplicação pode ser enganada, permitindo um acesso ao controle de fluxo o qual o atacante pode ver todos os livros, não somente os livros que combinam com o filtro permitido (mostrando livros publicados). Entretanto, descreveremos brevemente como um fluxo de injeção de SQL pode ser usado para extrair dados arbitrários de diferentes tabelas de bancos de dados e escalar privilégios dentro do banco de dados e do servidor de banco de dados. Por esta razão, qualquer vulnerabilidade de SQL injection deve ser levada extremamente a sério, independentemente do contexto da funcionalidade dentro da aplicação.

Injetando diferentes tipos de

instruções

A linguagem SQL contém um número de verbos que pode aparecer no começo das instruções. Por causa disto, o verbo mais comum, na maioria das injeções de SQL estão dentro da instrução *SELECT*. De fato, as discussões sobre SQL injection sempre dão a impressão que as vulnerabilidades ocorrem somente em conexões com instrução *SELECT*, porque os exemplos usados são todos deste tipo. Entretanto, vulnerabilidades de SQL injection podem existir dentro de qualquer tipo de instrução. Você precisa estar atento sobre algumas considerações importantes na relação de cada um deles. Claro que quando você está interagindo com alguma aplicação remota, isto normalmente não é possível você saber que tipo de instrução está sendo utilizada em um dado input do usuário para ser processado. No entanto, você geralmente pode educar sua percepção baseado nos tipos de funções da aplicação que você está lidando. A instrução SQL mais comum e seus usos serão vistos aqui.

Instrução SELECT

Instruções SELECT são usadas para obter informações do banco de dados. Eles são frequentemente empregados em funções onde a apliação retorna informação na resposta a uma ação do usuário, como navegar em um catálogo de produtos, ver informações do perfil do usuário ou realizando um busca. São sempre utilizadas em funções de login onde o usuário envia informações para serem checadas com o que for obtido em um banco de dados.

Como visto em um exemplo anterior, o ponto de entrada para ataques de SQL injection são normalmente na cláusula *WHERE* da consulta. Itens enviados pelo usuário são passados para o banco de dados controlar o escopo do resultado da consulta. Como o *WHERE* é normalmente um componente final de uma consulta *SELECT*, ele permite o atacante usar o símbolo de comentário

para truncar a consulta no final de seu input sem nenhuma validação de sintaxe de forma geral da consulta. Ocasionalmente, vulnerabilidades de SQL injection ocorrem quando afetam outras partes das consultas *SELECT*, como a cláusula *ORDER BY* ou os nomes das tabelas e colunas.

Instrução INSERT

Instruções *INSERT* são usadas para criar um novo registro de dados dentro da tabela. Eles são usadas normalmente quando uma apliação adiciona um novo registro em um log de auditoria, cria um novo usuário ou gera um novo pedido. Por exemplo, uma apliação pode permitir os usuários se registrarem, especificando seu usuário e senha, e pode então inserir os detalhes na tabela de usuários com a seguinte instrução:

INSERT INTO users (username, password, ID, privs) VALUES (‘daf’,‘secret’, 2248, 1)

Se o campo de usuário ou senha for vulnerável a SQL injection, um atacante pode inserir dados arbitrários na tabela, incluindo seu próprio valor de **ID** e **privs**. Entretanto, para fazer isto ele deve garantir que o restante da cláusula *VALUES* seja preenchido de forma correta. Em particular, ele deve conter o número correto de itens de dados com os tipos corretos. Por exemplo, injetando no campo *username*, o atacante pode fazer o seguinte:

foo’, ‘bar’, 9999, 0)--

Isto criará uma conta com o **ID** de **9999** e **privs** de **0**. Assumindo que o campo privs é usado para determinar os privilégios da conta, ele pode permitir o atacante criar um usuário administrativo.

Em alguns casos, quando estamos trabalhando totalmente as cegas, injetar em uma instrução *INSERT* pode permitir o atacante extrair dados da aplicação. Por exemplo, o atacante pode obter a versão do banco de dados e inserir no campo seu

próprio perfil de usuário, o qual pode ser exibido de volta no navegador de uma forma normal.

Quando estiver tentando injetar em uma instrução *INSERT*, você pode não saber quantos parâmetros são necessários ou quais os seus tipos. Na situação anterior, você pode continuar incrementando os campos da cláusula *VALUES* até que a conta desejada seja cadastrada. Por exemplo, quando injetamos no campo *username*, você pode fazer o seguinte:

```
foo')--
foo', 1)--
foo', 1, 1)--
foo', 1, 1, 1)--
```

Como a maioria dos bancos de dados implicitamente aceitando um inteiro como uma string, um valor inteiro pode ser usado neste ponto. Neste caso o resultado é uma conta com o usuário foo e a senha igual a 1, independentemente de qual é a ordem dos campos lá dentro.

Se você achar que o valor 1 está sendo negado, você pode tentar o valor 2000, que em muitos bancos de dados também aceitam implicitamente o tipo de dado baseado em data. Quando você determinar o número correto de campos seguidos do ponto de injeção, no MS-SQL você pode adicionar uma segunda consulta arbitrária e usar uma técnica baseada em inferência. No Oracle, uma consulta de subselect pode ser feita dentro da instrução de *INSERT*. Este subselect pode causar um sucesso ou falha na consulta principal, usando a técnica de inferência.

Instrução UPDATE

Instrução *UPDATE* são usadas para modificar um ou mais registros existentes nos dados de uma tabela. Eles sempre são utilizados quando um usuário modificar um valor que já existe. Por exemplo, atualizar seus dados de contato, alterando sua senha ou alterando a quantidade em uma linha de um pedido.

Uma instrução típica de *UPDATE* funciona como um *INSERT*, exceto que ele normalmente inclui a cláusula *WHERE* para dizer ao banco de dados quais linhas da tabela ele deve atualizar. Por exemplo, quando um usuário muda sua senha, a aplicação deve fazer a seguinte consulta:

```
UPDATE users SET password= 'newsecret' WHERE user = 'marcus' and password = 'secret'
```

Esta consulta verifica na tabela users onde o usuário e a senha esteja correta, para então poder atualizar para o novo valor. Se esta função for vulnerável a SQL injection, um atacante pode enganar a verificação da senha existente e atualizar a senha de um usuário administrador colocando o seguinte username:

```
admin'--
```

Procurar por vulnerabilidades de SQL injection em aplicações remotas é sempre perigoso, porque você não conhece qual ação a aplicação vai fazer se você utilizar algum input modificado. Em particular, modificar a cláusula *WHERE* em um *UPDATE* pode causar mudanças críticas através da tabela do banco de dados. Por exemplo, se um atacante enviar um username da seguinte forma:

```
admin' or 1=1--
```

Isto causaria na apliação a seguinte consulta:

```
UPDATE users SET password='newsecret' WHERE user = 'admin' or 1=1
```

Isto iria resetar o valor da coluna *password* de cada usuário, porque 1 é sempre igual a 1!

Tenha cuidado com este risco, mesmo que seu ataque seja em funções de aplicações que não pareçam atualizar qualquer dado existente, como a tela de login. Existem casos onde, após realizar um login com sucesso, a aplicação realiza várias atualizações utilizando o usuário fornecido. Isto significa

que qualquer ataque na cláusula *WHERE* poderá replicar em outras instruções, potencialmente causando estrago nos perfis de todos os usuários.

Você deve garantir que o dono da aplicação aceite estes riscos inevitáveis antes de tentar qualquer identificação ou exploração de vulnerabilidades de SQL injection. Você deve encorajar fortemente que o dono realize um backup completo antes de começar os testes.

Instrução DELETE

Instrução *DELETE* também é usada para deletar um ou mais registros dentro da tabela, como quando um usuário remove um item de seu carrinho de compras ou deleta um endereço de entrega de seu perfil. Assim como o *UPDATE*, uma cláusula *WHERE* normalmente é usada para dizer ao banco de dados quais linhas da tabela devem ser atualizadas. Os dados enviados pelo usuário são comumente incluídos nesta cláusula. Alterar a o destino da cláusula *WHERE* pode causar grandes danos e os mesmo cuidados do *UPDATE* devem ser tomados aqui.

Achando bugs de SQL injection

Na maioria dos casos óbvios, uma vulnerabilidade de SQL injection pode ser descoberta e conclusiva verificando através de uma entrada de dados inesperada em um item da aplicação. Em outros casos, bugs podem ser extremamente sutis e pode ser difícil de distinguir de outra categoria de vulnerabilidade ou do começo de uma anomalia que não é bem uma ameaça de segurança. No entanto, você pode realizar várias etapas de uma forma que seja confiável de verificar as vulnerabilidades deste tipo.

Injetando um string de dados

Quando um string de dados é enviado pelo usuário é incorporado em uma consulta SQL, ele é encapsulado dentro de uma marcação de aspas simples. Para explorar qualquer vulnerabilidade de SQL injection, você precisa quebrar estas marcações.

Etapas

1. Envie uma aspa simples como um string de dado que você esteja usando como alvo. Observe se acontecerá algum erro ou se o resultado é diferente do original de alguma forma. Se detalhes do erro do banco de dados for exibido, consulte o erro informado para entender o que de fato aconteceu.
2. Se um erro ou outro comportamente foi observado, envia duas aspas simples juntas. O banco de dados usa duas aspas simples como uma sequência de escape para representar literalmente uma aspa simples, então a sequência é interpretada como dados dentro da string marcada ao invés do caractere de fechamento. Se uma entrada causar erro ou comportamento diferente sumir, a aplicação é provavelmente vulnerável a SQL injection.
3. Como uma verificação aprofundada se o bug está presente, você pode usar o caracter concatenador de SQL para construir um string que é equivalente a uma entrada de dados benigna. Se a aplicação manipular sua entrada de dados trabalhada da mesma forma que faria normalmente, ela provavelmente é vulnerável. Cada tipo de banco de dados usa métodos diferentes de concatenadores. Veja os exemplos seguintes que podem ser injetados para construir uma entrada equivalente para FOO na aplicação vulnerável:
    1. Oracle: **‘||’FOO**
    2. MS-SQL: **‘+’FOO**
    3. MySQL: **‘ ‘FOO** (note o espaço entre as duas aspas)

Uma forma de confirmar se a aplicação está interagindo com um banco de dados é enviando o caractere coringa do SQL em um dado parâmetro. Por exemplo, enviando ele é um campo de pesquisa que sempre retorna um resultado grande, indica que a entrada de dados está sendo passada em uma consulta SQL. Claro, que isto não necessariamente indica que a aplicação é vulnerável – somente que você deve ir mais a fundo para identificar qualquer vulnerabilidade.

Enquanto estiver procurando por SQL injection usando aspa simpls, fique de olho se acontece qualquer erro no JavaScript quando o seu navegador interpretar o retorno do erro SQL. É comum em aplicações onde o usuário deve enviar dados para retornarem posteriormente ao JavaScript, e uma aspa simples não tratada devidamente pode causar um erro no interpretador do JS, assim como acontece com o interpretador do SQL. A possibilidade de injetar JavaScritp arbitrário em respostas permite realizar ataques chamados de Cross-Site Scripting (XSS).

Injetando dado numérico

Quando um dado numérico enviado pelo usuário é incorporado à consulta SQL, a aplicação pode manipular esta string de dados encapsulando-a dentro de uma marcação de aspa simples. Portanto, você sempre deve seguir os passos anteriores para string de dados. Em muitos casos, entretanto, dados numéricos são passados diretamente para o banco de dados em formato numérico e portanto não são colocados entre aspas simples. Se nenhum dos testes anteriores exibiram a possível presença de uma vulnerabilidade, você pode fazer os seguintes passos específicos para dados numéricos.

1. Tente fornecer uma expressão matemática simples que seja equivalente ao valor numérico original. Por exemplo, se um valor original é 2, tente enviar 1+1 ou 3-1. Se a aplicação responder da mesma forma, ele pode ser

vulnerável.

2. O teste anterior é mais confiável nos casos em que você tenha confirmado que o item a ser modificado tem um efeito perceptível sobre o comportamento do aplicativo. Por exemplo, se a aplicação usa um parâmetro numérico PageID para especificar qual conteúdo deve ser retornado, substituir 1+1 para 2 com resultado equivalente é um bom sinal de que SQL injection está presente. Entretanto, se você colocar uma entrada arbitrária em um parâmetro numérico sem mudar o comportamento da aplicação, o teste anterior não provê nenhuma evidência de vulnerabilidade.
3. Se o primeiro teste foi feito com sucesso, você pode obter mais evidências da vulnerabilidade usando expressões mais complexas com palavras-chaves SQL e sintaxe. Um bom exemplo é o comando ASCII, o qual retorna o valor numérico ASCII de um caractere. Por exemplo, porque o valor ASCII do A é 65, a expressão seguinte é equivalente a 2 em SQL: **67-ASCII('A')**
   4. O teste anterior não vai funcionar se a aspa simples estiver sendo filtrada. Entretanto, nesta situação você pode explorar o fato do banco de dados implicitamente converter dados numéricos em string de dados quando solicitado. Portanto, porque o valor ASCII do caractere 1 é 49, a expressão seguinte é equivalente a 2 em SQL:
   51-ASCII(1)

Um erro comum qunado estamos procurando por defeitos, como SQL injection, nas aplicações, é esquecer que certos caracteres tem um significado especial dentro das solicitações HTTP. Se você quer incluir estes caracteres dentro do payload de ataque, você deve ter o cuidado de codificar os dados para URL e garantir que eles serão interpretados da forma que você deseja. Em particular:

- **&** e **=** são usados para juntar pares de nomes/valores para criar uma string de consulta e o bloco de dados do POST.

Você deve codifica-los usando **%26** e **%3d**, respectivamente.

- Espaços literais não são permitidos nas strings de consultas. Se eles forem enviados, vão efetivamente terminar a string inteira. Você pode codifica-los usnado **+** ou **%20**.
- Como o + é usado para codificar espaços, se você quer incluir um caractere de + na sua string, você deve codifica-lo usando **%2b**. No exemplo numérico anterior, ao invés de usar 1+1, deveria ser enviado como **1%2b1**.
- O ponto-vírgula é usado para separar campos dos cookies e deve ser codificado usando **%3b**.

Estas codificações são necessárias se você está editando os valores do sparâmetros diretamente do seu navegador, com um proxy interceptando (Burp Suite), ou através de qualquer outro meio. Se você error ao codificar os caracteres, você pode invalidar a solicitação completa ou enviar dados que não tinha intenção de enviar.

Estes passos descritos são geralmente o suficiente para identificar a maioria das vulnerabilidades de SQL injection, incluindo muitos daqueles que retornam dados sem utilidades ou sem informações de erros que são transmitidos de volta para o navegador. Em alguns casos, entretanto, técnicas mais avançadas são necessárias, como o uso de *time delays* para confirmar a presença da vulnerabilidade.

Injetando na estrutura da consulta

Se o dado de um usuário é inserido na estrutura da consulta SQL em si, ao invés de então

Se os dados fornecidos pelo usuário está sendo inserido na estrutura da consulta SQL em si, em vez de um item de dados dentro da consulta, a exploração do SQL injection fica mais simples, fornecendo diretamente a sintaxe SQL válida. Não é necessário fazer o escape com aspa simples para quebrar

qualquer contexto de dados.

O ponto mais comum de injeção dentro da estrutura de consulta SQL está dentro da cláusula *ORDER BY*. O *ORDER BY* pega uma coluna ou um número e ordena o resultado de acordo com os valores contidos nesta coluna. Esta funcionalidade é frequentemente exposta, permitindo o usuário ordenar a tabela no navegador. Um exemplo típico de uma tabela ordenada é obtido através da seguinte consulta:

```
SELECT autor,titulo,ano FROM livros WHERE editora = 'XPTO' ORDER BY titulo ASC
```

Se o nome da coluna titulo do *ORDER BY* for especificado pelo usuário, não é necessário usar uma aspa simples. O dado do usuário é modificado diretamente na estrutura da consulta SQL.

Em alguns casos raros, a entrada de dados do usuário pode especificar o nome da coluna dentro da cláusula *WHERE*. Como estes não são encapsulados em aspa simples, problemas similares ocorrem aqui. Também é possível encontrar aplicações onde o nome da tabela é um parâmetro enviado pelo usuário. Finalmente, uma quantidade surpresa de aplicações expõe a palavra-chave de ordenação (*ASC* ou *DESC*) para ser definido pelo usuário, possivelmente acreditando que isto não terá consequência de um possível ataque de SQL injection.

Achar SQL injection no nome de uma coluna pode ser difícil. Se o valor informado não for válido como nome de coluna, a consulta resulta em um erro. Isto significa que a resposta será independentemente do que o atacante enviar como string de caminho transversal, aspa simples, aspa dupla ou qualquer outra string arbitrária.

Entretanto, técnicas comuns para automatizar os testes manuais e de fuzzing são viáveis para achar a vulnerabilidade. As strings padrões de testes para vários tipos de vulnerabilidades irão causar a mesma resposta, a qual pode não vazar a natureza do erro.

Usar prepared statements ou escapar com aspas simples não irão previnir este tipo de SQL injection. Como resultado, este vetor é a chave para olhar em aplicações modernas.

1. Tome nota de qualquer parâmetro que apareça para controlar a ordenação ou tipos de campos dentro do resultado que a aplicação retorna.
2. Faça uma séria de solicitações enviando parâmetros com valores numéricos, começando com o número 1 e incrementando nas consultas subsequentes:
    1. Se mudar o número na entrada de dados afetar a ordem dos resultados, o input provavelmente está sendo inserido na cláusula *ORDER BY*. Em SQL, **ORDER BY 1** ordena pela primeira coluna. Incrementar para o número 2 deve mudar a ordem como os dados estão sendo exibidos pela segunda coluna. Se o número informado for maior que o número de colunas da consulta, a consulta irá falhar. Neste caso, você pode confirmar injetando um SQL para reverter, usando o seguinte:
    ASC –
    DESC –
3. Se o número 1 informado causar uma série de número 1 no resultados na coluna, a entrada de dados está inserindo no nome da coluna e retornando na consulta. Por exemplo: SELECT 1,titulo,ano FROM livros WHERE editora = ‘XPTO’

Explorar SQL injection na cláusula *ORDER BY* é diferentemente significante da maioria dos casos. Um banco de dados não aceitará um *UNION*, *WHERE*, *OR* ou *AND* neste ponto da consulta. Geralmente, a exploração requer que um atacante defina uma consulta aninhada no lugar de um parâmetro, como substituindo o nome da coluna com **(select 1 where <<condition>> or 1/0=0)**, aproveitando assim as técnicas de inferência. Para banco de dados que suportam consultas em batch como MS-SQL, isto pode ser uma opção eficiente.

Fonte: The Web Application Hacker’s Handbook – 2nd edition.