第一章:**SQL**インジェクション

SQLインジェクションおよびJavaソースコードへの影響について学ぶ

SQLインジェクションとは？

Javaの場合

概要

SQL(構造化照会言語)インジェクションの攻撃は、クライアントからアプリケーションへの入力データにSQLクエリを挿入(インジェクション)して、アプリケーションに取り込まれることで成立します。SQLコマンドがデータプレーン入力に挿入されて、あらかじめ定義されているSQLコマンドの実行に影響を与えることになります。

開発者が検証やエンコードを行わないで、ユーザ入力されたデータが含まれるSQL文を作成する場合、SQLインジェクション攻撃の可能性があります。 このような攻撃の目的は、ユーザが通常アクセスできないデータをデータベースから取得したり、出力することです。ハッカーは、SQLインジェクション攻撃を使用して、機密性の高いビジネス情報や個人情報(PII)にアクセスします。そして、これが結果的に機密データの流失を広げます。

SQLインジェクション攻撃は、OWASPトップ10の脆弱性の中で最も一般的な脆弱性であり、最も古くから存在するアプ リケーションの脆弱性の一つです。ある最近の報告によると、3番目に多い深刻な脆弱性として挙げられています。

影響

SQLインジェクションの攻撃が成功すると、データベースからの機密データの読取り、データベースのデータの変更(挿入/更新/削除)、データベースでの管理操作の実行、DBMSファイルシステム上に存在する特定のファイルの内容の復元などが可能になりま

す。場合によっては、オペレーティングシステムに対してコマンドを発行することもできます。

例えば：攻撃者は脆弱なアプリケーションでSQLインジェクションを利用して、開発者が作成したクエリに顧客のクレジットカード番号などが含まれていなくても、そのようなデータをデータベースに問い合わせることができます。

どのように悪用されるのか？

攻撃者がSQLインジェクション攻撃を行うには、WebアプリケーションやWebページで脆弱な入力を見つける必要があります。アプリケーションやWebページにSQLインジェクションの脆弱性がある場合、ユーザのSQLクエリとして入力が直接使用されています。ハッカーが、悪意のあるサイバー攻撃として、特別に細工したSQLコマンドを実行できる可能性があります。そして、ハッカーは悪意のあるコードを活用することで、データベースの構造に関して明確に理解できるレスポンスを得て、それによってデータベース内の全ての情報にアクセスできるようになります。

開発者のためのSQLインジェクション – パート 1：SQLインジェクションの概要と説明

攻撃者は次のような方法で**SQL**インジェクションを行う可能性がある：

常に真**(true)**である**SQL**文。ハッカーは、常に真(true)となるSQL文でSQLインジェクションを実行します。例えば、1=1です。ハッカーは単に「間違った」入力をするのではなく、常に真となるステートメントを使用します。

クエリ入力欄に“100 OR 1=1”と入力することで、テーブルの詳細情報が含まれたレスポンスが返されます。

"OR ""="

このSQLインジェクションのアプローチは、上記の方法と似ています。この方法では、悪意のある第三者は、クエリ入力欄に"OR ""="を入力する必要があります。これら2つの符号は、アプリケーションに侵入するための悪意のあるコードとして機能します。

例えば、攻撃者がアプリケーションからユーザデータを取得しようとして、単にユーザIDやパスワードに“OR=”と入力するとします。このSQL文は有効で真(true)となるので、データベースにあるユーザテーブルのデータが返されることになります。

SQLインジェクションの種類

QLインジェクションは、インバンド、ブラインド、アウトオブバンドの3つに分類されます。

インバンドSQLインジェクション

最も頻繁に使用されるSQLインジェクション攻撃です。インバンド攻撃で使用されるデータの転送は、Web上のエラーメッセージによって行われるか、SQL文のUNION演算子を使用して行われます。

インバンドSQLインジェクションは2種類：UNIONベースのSQLインジェクション、エラーベースのSQLインジェクション

UNIONベースのSQLインジェクション：アプリケーションにSQLインジェクションの脆弱性があり、アプリケーションのレスポンスにクエリの結果が返される場合、攻撃者はUNIONキーワードを悪用して、アプリケーションのテーブルだけでなく他のテーブルからデータを取得することが可能になります。

エラーベースのSQLインジェクション：エラーベースのSQLインジェクションは、アプリケーションのデータベースサーバが出力するエラーメッセージに依存する方法です。これによって、攻撃者はエラーメッセージの情報を利用して、データベースの実体を特定することができます。

ブラインドSQLインジェクション

攻撃者がデータペイロードを送信した後に、その動作やレスポンスを観察して、データベースのデータ構造を特定する攻撃です。

ブラインドSQLインジェクション(または推論SQLインジェクション)は2種類：ブール型ベース、時間ベース

ブール型ベース(Boolean-based)：ブール型ベースの手法では、SQLクエリをデータベースに送信して、アプリケーションにブール型の結果、つまりTUREか

FALSEのどちらかの結果を返させます。攻撃者は、様々なクエリをやみくもに実行し、脆弱性を探り出します。

時間ベース**(Time-based)**：時間ベースのSQLインジェクション攻撃は、アプリケーションが一般的なエラーメッセージを返す場合によく使われます。この手法では、一定時間データベースを待機させます。応答時間によって、攻撃者はクエリの戻り値がTUREかFALSEかを判別することができます。

アウトオブバンド**SQL**インジェクション

アウトオブバンドSQLインジェクション攻撃では、アプリケーションがHTTP、DNS、SMBなどの任意のプロトコルでデータを送信するようリクエストします。この種の攻撃を実行するには、Microsoft SQLデータベースとMySQLデータベースでそれぞれ以下の関数を使うことができます。

MS SQL: master..xp _dirtree
MySQL: LOAD_FILE()

JavaでのSQLインジェクション

SQLインジェクション攻撃を阻止するのに最も効果的な方法は、データベースとのやり取りを安全に処理するHibernateのようなマッピング(ORM)のみを使用することです。

クエリを直接操作する必要があるなら、ストアドプロシージャの場合はCallableStatementを、通常のクエリの場合はPreparedStatementを使用してください。これらのアプリケーションプログラミングインターフェイス(API)はどちらもバインド変数を使用します。どちらの手法も、適切に使用すれば、コードのインジェクションを完全に阻止できます。

ユーザ入力をクエリに連結することは避け、バインドパターンを使用してユーザ入力がSQLコードとして誤って解釈されないようにする必要があります。

以下は、安全でないクエリの例です。

```java
String user = request.getParameter("user");
String pass = request.getParameter("pass");
```

```
String query = "SELECT user_id FROM user_data WHERE user_name = '" + user + "' and
user_password = '" + pass +"'";
try {
 Statement statement = connection.createStatement(
ResultSet results = statement.executeQuery( query ); // 安全ではありません！
}
```

では、`PreparedStatement`を使って、上記のクエリを安全なものにしてみましょう。

```
Java
String user = request.getParameter("user");
String pass = request.getParameter("pass");
String query = "SELECT user_id FROM user_data WHERE user_name
= ? and user_password = ?";
try {
  PreparedStatement pstmt = connection.prepareStatement( query );
  pstmt.setString( 1, user );
  pstmt.setString( 2, pass );
  pstmt.execute(); // 安全です！
}
```

動的な検索のように、変数の順序や数が事前に決定できないために、パラメータ化されたクエリを使用するのが難しい場合もあります。

このようなSQLの呼び出しを動的に生成するのを避けられない場合は、全てのユーザデータの検証とエスケープ処理が必要です。

エスケープする文字は、使用しているデータベースと、信頼できないデータを入れるコンテキストにより異なります。

これを独自に行うのは大変ですが、幸いESAPI(OWASP Enterprise Security API)ライブラリにこのようなエスケープ機能が用意されています。

以下の例では、Oracleデータベースで、信頼できないデータを使用して動的に生成される構文を安全にエンコードしています。

```
Java
Codec ORACLE_CODEC = new OracleCodec();
String user = req.getParameter("user");
String pass = req.getParameter("pass");
String query = "SELECT user_id FROM user_data WHERE user_name = '" +
  ESAPI.encoder().encodeForSQL( ORACLE_CODEC, **user**) +
```

```
"' and user_password = '" +
 ESAPI.encoder().encodeForSQL( ORACLE_CODEC, **pass**) +
"'";{ {/javaBlock} }
```

MyBatisフレームワーク

動的SQLクエリで「${}」という表記を使用する場合、MyBatisは文字列の修正やエスケープを行いません。

このため、マップされた値がクエリに直接挿入され、SQLインジェクション攻撃につながる可能性があります。

MyBatisを使用するアプリケーションでは、信頼できないデータに対してバインド変数「#{}」を使用すべきです。

これを使用すると、MyBatisで文字列置換が作成されます。この文字列置換は、実行時にユーザ入力によって置き換えられるプレースホルダのある不完全なSQLクエリです。これにより、ユーザ入力はSQLコマンドの一部としてではなく、パラメータの内容として扱われます。

セカンドオーダーSQLインジェクション

悪意を持って細工された入力により、エンドユーザがSQLクエリの構造を変更し、実行時に直接実行するわけではなく、2次的(セカンドオーダー)にSQLインジェクション攻撃を実行する可能性があります。

セカンドオーダーSQLインジェクションは、ユーザが指定したデータがアプリケーションで格納され、後に安全でないSQLクエリにそのデータがトリガーされて、クエリ含まれる場合に可能になります。

このような攻撃の目的は、ユーザが通常アクセスできないデータをデータベースから取得したり、出力することです。例えば、攻撃者は、安全でないユーザ名を登録することによって、脆弱なWebアプリケーション上でセカンドオーダーSQLインジェクションを使う可能性があります。これが、ユーザテーブルに格納され、後でデータを取得したり操作する際に実行されることになります。

影響

セカンドオーダーSQLインジェクション攻撃が成功すると、データベースから機密データを読み取ることができます。さらに、権限昇格やアカウントの乗っ取りも可能となり、場合によっては、攻撃者がデータベースサーバへシェルアクセスできる可能性もあります。

対策

セカンドオーダーSQLインジェクション攻撃を阻止するのに最も効果的な方法は、データベースとのやり取りを安全に処理するHibernateのようなマッピング(ORM)のみを使用することです。クエリを直接操作する必要があるなら、ストアドプロシージャの場合はCallableStatementを、通常のクエリの場合はPreparedStatementを使用してください。これらのAPIではどちらもバインド変数を使用します。どちらの手法も、適切に使用すれば、コードのインジェクションが完全に阻止されます。ユーザ入力をクエリに連結することは避け、バインドパターンを使用してユーザ入力がSQLコードとして誤って解釈されないようにする必要があります。

以下は、安全でないクエリの例です。

```
Java
String user = request.getParameter("user");
String pass = request.getParameter("pass");
String query = "SELECT user_id FROM user_data WHERE user_name = '"
+ user + "' and user_password = '" + pass +"'";
try {
Statement statement = connection.createStatement( );
}
ResultSet results = statement.executeQuery( query ); // 安全ではありません！
```

では、PreparedStatementを使ってこれを修正しましょう。

```
Java
String user = request.getParameter("user");
String pass = request.getParameter("pass");
String query = "SELECT user_id FROM user_data WHERE user_name = ? and user_password
= ?";
try {
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, user );.setString( 2, pass );
pstmt.execute(); // 安全です！
}
```

動的な検索のように、変数の順序や数が事前に決定できないために、パラメータ化されたクエリを使用するのが難しい場合もあります。
このようなSQLの呼び出しを動的に生成するのを避けられない場合は、全てのユーザデータの検証とエスケープ処理が必要です。
エスケープする文字は、使用しているデータベースと、信頼できないデータを入れるコンテキストにより異なります。これを独自に行うのは大変ですが、幸いESAPIライブラリにこのようなエスケープ機能が用意されています。

以下の例では、Oracleデータベースで、信頼できないデータを使用して動的に生成される構文を安全にエンコードしています。

```
Java
Codec ORACLE_CODEC = new OracleCodec();
String user = req.getParameter("user");
String pass = req.getParameter("pass");
String query = "SELECT user_id FROM user_data WHERE user_name = '"
+ ESAPI.encoder().encodeForSQL( ORACLE_CODEC, **user**) +
"' and user_password = '"
+ ESAPI.encoder().encodeForSQL( ORACLE_CODEC, **pass**) + "'";
```

おつかれさまでした！

JavaでのSQLインジェクションとは何であるか、そしてシステムをSQLインジェクションから守るにはどうすればよいのかが、お分かり頂けたと思います。ここで学んだ新しい知識をうまく応用しながら、コーディングをしてください。これをあなたのネットワークで自由に共有してください。また、一般的な脆弱性に関する弊社のその他のレッスンも是非ご覧ください。
この学習モジュールに関して修正などがある場合は、こちらをクリックして、プルリクエストを作成してください！

関連記事：

ブログ：Why SQL Injection Attacks Still Need to be Dealt With

ブログ：The Top 10 app-attack trends in the financial sector in 2022