SQL Injection Attacks

CMPU4023 - Enterprise Application Development

Query Execution

- Recall that, from the application programmer's perspective, queries are sent to the database in the form of SQL language statements and results are returned in the form of rows

- The database will execute whatever query is presented and has permissions to run so it is the responsibility of the programmer to ensure that the queries are ultimately safe

Dynamic Generation

- In most cases the SQL query statements (strings) are dynamically generated by the middleware before being forwarded to the database
- The SQL statements may need to include input from the client to qualify the query such as a string match or a numeric limit or similar
- The user-input element is what gives rise to the potential for one of the most serious security vulnerabilities in service software - a SQL injection attack
- In principle, no user input can ever be trusted by the server
- Badly implemented query generation that does not heed this can be easily compromised

Exposure

- A successful attack can have a devastating impact on the business of the enterprise
- The attacker can get access to user account information such as passwords or personal and financial details or gain the same privileges as a user
- The attacker could also change or delete information such as email addresses or passwords or drop entire tables
- In many cases the intrusions may go completely undetected which deepens the exposure to the business

Pathology

- What the attacker wants to do is exploit a query formation vulnerability by repeatedly sending queries to the service with malformed input in the hope that he will find a flaw
- Once a flaw is found, the attacker can potentially mount arbitrary attacks on the system to learn more about the schema, the data and extract or modify critical values
- Let’s consider how this can be done using a number of examples

Credentials Vulnerability

- Suppose that a user login check query string was built as a string as follows:

```
const q = `
    SELECT field-list
    FROM users u
    WHERE u.email = '${email}'
    AND u.passwd_hash = crypt('${passwd}', u.passwd_hash);
`;
```

- This JS code example is using the ES6 templating feature to insert field values for email and password

Credentials Exploit

- Suppose the attacker was able to craft values as follows:

email	passwd
`x' OR 'x' = 'x`	`x','y') OR 1 = 1; --`

- When our query string is expanded then it will look like the following:

```
const q = `
    SELECT field-list
    FROM users u
    WHERE u.email = 'x' OR 'x’ = 'x'
    AND u.passwd_hash = crypt('x','y') OR 1 = 1; --', u.passwd_hash);
`;
```

Update Vulnerability

- Suppose that a table update string is built as a string as follows:

```
const q = `
    UPDATE products SET
        products.name = '${name}'
    WHERE products.id = '${id}';
`;
```

- Again, the name and id parameters are added directly into the query string

Update Exploit

- Suppose the attacker was able to craft a value as follows:

name
`' || products.name, products.price = 0.0; --`

- When our query string is expanded then it will look like the following:

```
const q = `
    UPDATE products SET
        products.name = '' || products.name, products.price = 0.0; --'
    WHERE products.id = '${id}';
`;
```

Eliminating the Vulnerabilities

- The fundamental problem with generating SQL query strings this way is that the attacker has the opportunity to inject the bad code before the query planner gets a chance to parse it
- The fix is to generate the queries a different way by:

  - Using pre-parser function which checks the validity of the query before it is executed (e.g. `mysqli::escape_string` from PHP) - not as secure in practice
  - Using prepared statement or parameterised queries
  - Using a stored procedure with typed arguments
  - Isolating the execution within a tight security sandbox using database privileges

Prepared Statements

- Using a prepared statement, the middleware builds and the databases parses the query string ahead of execution time
- Parameter values are provided at the execution time and are never therefore parsed as code

```
const q = `
    SELECT field-list
    FROM users u
    WHERE u.email = $1
    AND u.passwd_hash = crypt($2, u.passwd_hash);
`;
```

- The query parameters for email and password are supplied at execution time and not at query build time

Stored Procedures

- Stored procedures offer the same ahead-of-time parsing and runtime parameter type-checking facility

```
CREATE OR REPLACE FUNCTION authenticate(_email TEXT, _passwd TEXT)
RETURNS uuid
AS $$
    SELECT u.id
    FROM users u
    WHERE u.email = _email
    AND u.passwd_hash = crypt(_passwd, u.passwd_hash);
$$ LANGUAGE SQL STRICT IMMUTABLE;
```

- Marking it as immutable add extra safety indicating that no database updates are permitted

Database Privileges

- Modern RDBMSs have sophisticated authorisation features which can allow very fine grained access to database resources
- A partial mitigation against injection attacks and as general good practice is to revoke all unnecessary privileges from the query execution user and only grant the particular privileges necessary, including setting up different users for different kinds of queries and tables

Summary

- SQL injection attacks are a security exploit of vulnerable query generation practices in code
- If the attacker can insert values of his choosing into dynamically generated query strings then he can execute arbitrary code in the database which could read or update business-sensitive data
- Attacks on databases can be devastating for business continuity and reputation
- Programmers should use best practices when constructing queries such as prepared statements or stored procedures

Refrences

- https://phpdelusions.net/sql_injection