Web Penetration Testing and Ethical Hacking

SANS

Exploiting esoteric SQL injection vulnerabilities

Bojan Zdrnja | bojan.zdrnja@infigo.hr | @bojanz on Twitter

Agenda

- $ whoami
- SQL injection 101
  - Just a quick refresher
- In-band or inline and Blind SQL injection vulnerabilities
- Determining true vs false cases
- A (relatively) simple demo
- Two esoteric examples demo
  - Nevertheless found in the wild

$ whoami

- Bojan Ždrnja
- Senior Information Security Consultant at INFIGO IS
  - http://www.infigo.hr
  - Penetration testing team lead
- SEC542 and SEC504 instructor
- SANS Internet Storm Center handler
  - https://isc.sans.edu

- Twitter: @bojanz

SQL injection 101

- Injection vulnerabilities are still the most common vulnerabilities in web applications today
  - Holding the #1 spot in OWASP Top 10 since 2010
- OWASP definition

  "*Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query.*

  The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization."

SQL injection 101

- SQL injection vulnerabilities due to interaction of a web application with a relational database
  - The application needs to read, update, insert or delete data from the database
- Interaction with the database often depends on user activities
  - As such a user's input will be passed to the database sooner or later
  - This data will be parsed by the interpreter as part of a query
  - The attacker can trick the interpreter into executing arbitrary commands and access data in the database

SQL injection 101

- Virtually all web applications use a single account to access the database
  - Desirable behavior for better performance
    - Especially with connection pooling
- This means that the web application must handle authorization
  - The account used to connect to the database can typically access all tables
    - Rarely we see multiple accounts used for database connections
  - If an attacker circumvents the web application's authorization he can retrieve any data
    - Or write arbitrary data provided the SQL query allows that

SQL injection 101

- Important SQL verbs
  - SELECT (read from a table)
  - INSERT (write to a table)
  - UPDATE (modify table contents)
  - DELETE (remove rows from a table)
  - UNION (combines the results of two or more SELECT statements into one)
- Important SQL modifiers
  - WHERE (filter query to apply only to the condition specified)
  - AND/OR (combine with WHERE to narrow the SQL query)

SQL special characters

- Used for various purposes
  - String delimiters: `'` and `"`
  - Comment delimiters: `--` `#` `/* */`
  - String concatenation characters: `||` `+` `' '`
  - Mathematical operators: `+` `<` `>` `-`
  - Calling functions `( )`
    - Very important for subqueries
    - I.e. `( SELECT 123 )`

SQL injection 101

- A very basic example:

```
$sql = 'SELECT * FROM transactions WHERE source = \'' . $_GET['id'] . '\';';
```

- Contents of the `id` parameter are inserted directly into the SQL query
- SQL's most dangerous character: `'`
  - The SQL query becomes invalid

```
SELECT * FROM transactions WHERE source = '' ';
```

Our injected character

SQL injection 101

- Extending the query to perform arbitrary actions
  - Injecting the famous `' OR '1'='1`
  - The SQL query becomes this:

```
SELECT * FROM transactions WHERE id = '' OR '1'='1' ';
```

Our injection resulted in valid SQL

  - Thanks to the all powerful SQL interpreter we can extract any data now

Blind SQL injection

- Blind SQL injection occurs when we do not see the result of a query directly
  - If an error happened, we cannot see what the error was
  - Good web applications will always display a generic screen when an error is encountered
  - However, we know that something has happened in the background
- This can also be **easily** exploited
  - Just requires a bit more ingenuity
- In this case we have to infer what the result of the modified query was

SQL injection 101

- What is the outcome of the following requests
  - http://site/press.php?releaseID=1
    - Works
  - http://site/press.php?releaseID=a
    - We get a blank screen (could be due to an error?)
  - http://site/press.php?releaseID=2
    - Works (as expected)
  - http://site/press.php?releaseID=2-1
    - Works! This potentially indicates a blind SQL injection vulnerability
  - http://site/press.php?releaseID=10-9
    - Works as well, and we get the same result as the first query

SQL injection 101

- We know that an interpreter parsed our injection
- Since we can influence the SQL query our next step is to distinguish between the *true* and *false* cases
  - We ask database a question
  - Determine the answer based on the response
- We have to guess one character at a time
  - In reality we can speed this up through clever optimization
    - This is a topic for another webcast

SQL injection 101

- By extending the SQL query we can yield *true* and *false* cases
  - *True*

```
SELECT * FROM transactions WHERE id = ' ' AND '1'='1  ';
```

  - *False*

```
SELECT * FROM transactions WHERE id = ' ' AND '1'='2  ';
```

  - Guess the first character of the username used by the web application to connect to the database

```
SELECT * FROM transactions WHERE id = ' ' AND (SELECT LOWER(SUBSTR(USER(),1,1))) = 'a ';
```

True if the first character is ‘a’

Demo time

- Let's see two simple demos
- In both cases we are dealing with a blind SQL injection vulnerability
- We will be using the sqlmap tool to test
- For manual tests we will use Burp Suite Professional
  - **Both tools are covered in SEC542**
- Case #1:

```
SELECT * FROM transactions WHERE id = ' $id ';
```

Demo time

- Case #1.1:

```
SELECT * FROM transactions WHERE id = $id;
```

- Notice there are no quotes around the `id` parameter
  - Not needed since it is a numerical parameter
    - Even this can cause problems for some scanners ☹
- Case #2:

```
SELECT * FROM transactions WHERE id = $id+10000;
```

- The developer adds a base value to the `id` parameter
  - And verifies in the code if `id` is numerical

Demo time

- Additional check by the developer

```
if ($result->num_rows > 0) {
// output data of each row
        while($row = $result->fetch_assoc()) {
                if (is_numeric($row['source']) and ($id+5 == $row['source'])) {
                        echo "Source: " . $row['source'] . ", Destination: " . $row['destination']
                        . ", Amount: " . $row['amount'] . ", Date: " . $row['datetime'] . "<br>";
                }
        }
} else {
        echo "No transactions found.<br>";
}
```

- Verifies if the results belong to the `id` submitted
  - A minor obstacle for scanners

Fingerprinting the database

- ORACLE
  - `'Concat' || 'enation'`
  - `BITAND(1,1)-BITAND(1,1)`
- MS-SQL, Sybase
  - `'Concat' + 'enation'`
  - `@@PACK_RECEIVED-@@PACK_RECEIVED`
- MySQL
  - `'Concat'  'enation'`
  - `CONNECTION_ID()-CONNECTION_ID()`

Ramping up the game

- Now for a more complicated example
  - The next vulnerability is not detected by almost any automated scanner
    - At least not with those I tried, including sqlmap as we will see in a minute
- In this example we have a developer that blacklisted certain keywords
  - SLEEP, BENCHMARK, UNION, CASE, AND, MAKE, ELT and --
  - These keywords are typically used by scanners to identify SQL injection vulnerabilities
    - SLEEP and BENCHMARK for timing attacks on MySQL
      - On MSSQL WAITFOR DELAY is used

Ramping up the game

- The query is the same as before

```
SELECT * FROM transactions WHERE id = $id;
```

- Remember we can test existence of SQL injection vulnerabilities with mathematical operators
  - `id = 1002-2`
  - `id = 10000-9000`
- As well as with subqueries
  - `(SELECT 1000)`
- These all will return the same result

Ramping up the game

- We need to guess one character at a time
  - We will be asking the application many questions
- There are multiple ways to exploit this
  - In this example `id=1000` returns a valid page
    - This will be "*true*"
  - Any other id value either returns a different page or nothing
    - This will be "*false*"

Ramping up the game

- We need to cycle through ASCII characters
  - A = 65 decimal
  - z = 122 decimal
- Our injection

```
(SELECT 1065 - (SELECT ASCII(LOWER(SUBSTRING(USER(),1,1))))
```

Cycle this from 1065 to 1222

- Once we guess the value of the first character we get back the web page for `id=1000`

Ramping up the game

- We have our *true* and *false* cases
- Now it's all up to scripting
- Demo time!

Seemingly impossible?

- What when seemingly there is no *true* or *false* web page?
  - For example, we are testing a new application
    - And there is no data in the database yet!
      - Never happened, right?
    - Every query returns an empty screen
    - This is also missed by most (all?) web scanners
- The first step is to determine if there is SQL injection:
  - Modify the input to deliberately break the SQL query
    - I.e. by inserting `'` characters
  - Modify the input to contain valid SQL query

Seemingly impossible?

- Our seemingly impossible injection

```
SELECT * FROM transactions WHERE mobile IN ( ' . $type . ' ) AND source = ' . $id . ';
```

- The `id` parameter is non-injectable
- The `type` parameter must have literal values `true` or `false`
- Again the developer blacklisted some keywords
  - SLEEP, BENCHMARK, UNION, LIKE, OR
- The main issue is in determining *true* and *false* cases
  - Instead of displaying different content, try causing an error

Seemingly impossible?

- We need to deliberately cause a SQL error (exception)
- Normally we can use "Division by zero error"
  - Simply add `( SELECT 1/0 )`
  - Does not work with MySQL (does not raise an exception)
- With MySQL we do have an option
  - Retrieve a column from a table that has multiple rows
    - Must be a subquery
    - Will cause exception
      - `ERROR 1242 (21000): Subquery returns more than 1 row`

Seemingly impossible?

- Luckily we have such a table readable by normal, non privileged accounts
  - `(select table_name from information_schema.tables)`
- Returns all tables in the database
- We need to modify the content of the `type` parameter
  - Our *true*/*false* case scenario will be this:
    - For *true*, we will cause a "`Subquery returns more than 1 row`" error
    - For *false*, the `type` parameter must hold the string "`true`"

Seemingly impossible?

```
'tr' +
(SELECT IF(ASCII(LOWER(SUBSTR(USER(),1,1)))=65,
    (SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES),
    'u'))
+ 'e
```

Cycle this from 65 to 122

- If we did not guess the character, the string concatenates to:
  - `tr' + 'u' + 'e`
- If we guessed it, cause a '`Subquery returns more than 1 row`' error
- Demo time!

Lessons learned

- We have seen examples of seemingly unexploitable SQL injection vulnerabilities
  - These were some extreme examples
    - But all based on real world cases!
  - Still, with some cleverness we managed to exploit them
- We cannot 100% rely on scanners
  - They are good, but they can miss some edge cases
  - And a vulnerable endpoint is enough for an attacker

Lessons learned

- Developers should be aware of this
  - Blacklisting never works, whitelisting should be always used
  - Best way to prevent SQL injection vulnerabilities is to use prepared statements (parametrized queries)
    - Do not build SQL queries by directly concatenating/joining strings
  - All user input should be properly escaped/filtered
    - Both on the client and the server sides
  - Enforce least privilege access on databases
    - User account used by a web application should have minimal privileges required for normal operation

Thank you for attending!

- Questions: bojan.zdrnja@infigo.hr / @bojanz